Green AI: Do Deep Learning Frameworks Have Different Costs?

Stefanos Georgiou*
Queen's University
stefanos.georgiou@queensu.ca

Maria Kechagia*
University College London
m.kechagia@ucl.ac.uk

Tushar Sharma
Dalhousie Uninversity
tushar@dal.ca

Federica Sarro
University College London
f.sarro@ucl.ac.uk

Ying Zou
Queen's University
ying.zou@queensu.ca

ABSTRACT

The use of Artificial Intelligence (AI), and more specifically of Deep Learning (DL), in modern software systems, is nowadays widespread and continues to grow. At the same time, its usage is energy demanding and contributes to the increased CO_2 emissions, and has a great financial cost as well. Even though there are many studies that examine the capabilities of DL, only a few focus on its *green aspects*, such as energy consumption.

This paper aims at raising awareness of the costs incurred when using different DL frameworks. To this end, we perform a thorough empirical study to measure and compare the energy consumption and run-time performance of six different DL models written in the two most popular DL frameworks, namely PyTorch and TensorFlow. We use a well-known benchmark of DL models, DeepLearningExamples, created by NVIDIA, to compare both the training and inference costs of DL. Finally, we manually investigate the functions of these frameworks that took most of the time to execute in our experiments.

The results of our empirical study reveal that there is a statistically significant difference between the cost incurred by the two DL frameworks in 94% of the cases studied. While TensorFlow achieves significantly better energy and run-time performance than PyTorch, and with large effect sizes in 100% of the cases for the training phase, PyTorch instead exhibits significantly better energy and run-time performance than TensorFlow in the inference phase for 66% of the cases, always, with large effect sizes. Such a large difference in performance costs does not, however, seem to affect the accuracy of the models produced, as both frameworks achieve comparable scores under the same configurations. Our manual analysis, of the documentation and source code of the functions examined, reveals that such a difference in performance costs is under-documented, in these frameworks. This suggests that developers need to improve the documentation of their DL frameworks, the source code of the functions used in these frameworks, as well as to enhance existing DL algorithms.

*The first two authors contributed equally to this work.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS CONCEPTS

• **Hardware** → **Power and energy**; • **Software and its engineering** → **Software libraries and repositories**; • **Computing methodologies** → **Machine learning**;

KEYWORDS

Energy consumption, run-time performance, deep learning, APIs

ACM Reference Format:
Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. 2022. Green AI: Do Deep Learning Frameworks Have Different Costs?. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep learning (DL) is a field of machine learning (ML) that has recently gained significant attention from researchers and practitioners. Along with the increase of computational power and availability of data, the use of deep learning has contributed to the improvement of several applications (*e.g.*, in the medical, financial, transportation sectors) that, for instance, use speech and image recognition, machine translation, and natural language processing (NLP). The advancement in these areas would have not been possible without the great advancement in DL.

While the research community has spent a significant effort towards improving the accuracy of DL approaches, it has often overlooked their costs. As recently reported by Schwartz et al. [76], DL has been assisting in an increase in the computational costs of the state-of-the-art AI research as big as 3000,000x between 2012 and 2018. Such a dramatically increasing trend in resource consumption, dubbed as Red AI, is not just often prohibitively expensive for researchers and practitioners, but also environmentally unfriendly.

This has motivated the field of Green Software Engineering (SE) research, which aims to decrease software environmental footprints and supports, *inter alia*, Green AI [58]. Optimizing resource utilization used by expensive DL models, without compromising their accuracy, is important to combat such an environmentally unfriendly and prohibitively expensive trend [78].

This paper presents an in-depth empirical analysis to investigate and compare the energy consumption and run-time performance of DL frameworks, in particular, PyTorch and TensorFlow. To the best of our knowledge, this is the first such empirical study. We select six large models for DL from different AI domains (recommender systems, NLP, and computer vision) extracted from a popular benchmark *i.e.*, DeepLearningExamples [5]. We use the

benchmark to perform training and inference experiments involving DL algorithms (*e.g.*, Transformer, CNN, GNMT), and measure the consumed energy and run-time performance. Furthermore, to investigate issues that may affect the energy consumption and run-time performance of DL models, we examine whether particular functions provided within the PyTorch and TensorFlow APIs can affect the efficiency of DL models. Finally, we discuss potential suggestions for improving the energy consumption and run-time performance of DL models without compromising accuracy.

Our findings show that TensorFlow outperforms PyTorch regarding energy consumption by 1.7x and run-time performance by 2.1x when training models belonging to the recommender systems and computer vision categories of our benchmark. By contrast, PyTorch is 2.1x more energy-efficient and 2.4x faster than TensorFlow when training models of the NLP category. In the inference of the models, TensorFlow is 1.4x faster and 1.7x more energy-efficient than PyTorch only for the recommender systems and ResNet-50 models. Regarding accuracy, both frameworks achieve a similar score under the same configurations. Therefore, under our configurations, TensorFlow is overall more energy and run-time efficient compared to PyTorch in the training phase, but less efficient in the inference phase. Finally, we manually investigated the functions of the frameworks examined that took most of the time to execute in our experiments. We argue that the energy consumption and run-time performance of such functions could be improved, in the future, by both improving the documentation and source code of the DL frameworks, and optimizing existing core DL algorithms. The paper makes the following contributions.

(1) A publicly available framework, PrEngDL (it stands for Performance and Energy of Deep Learning) [6, 8].
(2) An empirical study to compare DL frameworks (PyTorch [69] and TensorFlow [10]) regarding energy and run-time efficiency using six large models for DL.
(3) An analysis of the issues that hinder the energy consumption and run-time performance of DL models, for PyTorch and TensorFlow, as well as a discussion on potential recommendations to mitigate the identified issues, in the future, without sacrificing accuracy.

2 BACKGROUND

Deep Learning. DL, a subfield of ML, allows computational models to compose and arrange in multiple processing layers to learn representations of data with multiple levels of abstraction [33, 53]. DL techniques are extensively used to solve a variety of detection, prediction, and classification problems belonging to various domains. These domains include image recognition [47, 80], speech recognition [75], and NLP [43]. As in traditional supervised ML, DL models consist of two processes: *training* and *inference*. Training refers to the process of learning weights of the internal nodes of a DL model using training data, to *learn* patterns from the data. Inference refers to the process of using a trained DL model to make a prediction on *unseen* data.

Each layer in a DL model transforms the sequence of data coming from the previous layer or learns the representation of the input data in the form of weights of the nodes. A well-designed DL model is capable of inferring features during training and can learn to classify samples based on these inferred features. For example, a Convolution Neural Network (CNN) mimics the alternating layers of simple and complex cells of the visual cortex in animals [29]. CNN-based DL models have been proven effective for image classification and detection [47, 80] and face recognition [52, 68]. Similarly, a Recurrent Neural Network (RNN) and a fully-connected Neural Network are among the various layers that are used to compose a DL model; their arrangement within a DL model is often influenced by the problem at hand [50].

The wealth of available software-specific artifacts coming from abundant open-source repositories and the advancements in the ML applications, beyond audio and images, have paved the way for a rapid growth of ML techniques for software engineering applications. To this end, the repetitive and predictable nature of the source code revealed by the statistical characteristics of the source code has been compared with the properties of the natural text [28, 38]. Also, researchers have extensively applied ML techniques for clone detection [85, 86], de-obfuscation [82], language migration [62], code summarization [41], auto-correction [34, 73], auto-completion [30], code generation [56, 63, 88], and program comprehension [11].

Energy and Run-time Efficiency. A programming task's (DL model here) energy consumption is the amount of energy, measured in *Joules*, required by a computer system to accomplish the task; the energy consumption is calculated by the formula $E = P \times T$ [51], where P denotes the power consumption, in *Watts*, and T the total amount of time (*run-time performance*), in *seconds*, required to execute a task. Although in the physical sense energy cannot be consumed, we will use the term *energy consumption* to refer to the conversion of electrical energy by IT equipment into thermal energy dissipated to the environment. For example, we say that a program A consumes 30 Joules and needs ten seconds to accomplish a task and program B consumes 20 Joules and needs 20 seconds to accomplish the same task as program A. Then, program A is more run-time efficient, while program B is more energy efficient. Additionally, as computer systems change and become more complex with an evolving memory hierarchy, having processors of multiple cores and distinct power states, power requirements begin to vary. Even though there are several studies related to computer systems and energy efficiency, it is still unclear how certain design decisions can alter the energy consumption of computer programs [16], or what trade-offs exist [58, 65].

3 EXPERIMENTAL SETUP

3.1 Research Questions

RQ1: Which is the most energy and run-time efficient DL framework for the models examined?—With RQ1, we examine the energy consumption and run-time performance of PyTorch and TensorFlow for the models used in our study. Knowing this information, we can inform the DL community about the most efficient frameworks for the models under evaluation.

RQ2: How accurate are the DL frameworks for the models under examination?—With RQ2, we examine the accuracy in the results produced by the DL frameworks for the models of our study, *i.e.*, we wish to see whether the selected frameworks sacrifice accuracy over energy or run-time efficiency, under our configurations. Knowing this information, we can inform the DL

community about the most accurate as well as energy and run-time efficient frameworks for a given model.

RQ3: What are the least energy and run-time efficient APIs of DL frameworks for the models under examination?—With RQ3, we investigate the reasons that make DL frameworks less energy and run-time efficient. In particular, we identify APIs in DL frameworks that are energy-hungry and run-time inefficient. Locating these APIs, we will be able to make suggestions for improving those APIs, in the future, and, thereby, the DL frameworks.

3.2 Benchmark

Benchmark. To achieve high accuracy, DL algorithms must learn differentiating patterns with the help of large data sets. We considered the following criteria to select an appropriate benchmark for our study.

(1) The benchmark should be *public*, *popular* (having a significant number of stars, which is a proxy of popularity [12]; we consider benchmarks with more than 100 stars on GitHub) and *maintained* (having recent commits on GitHub, *i.e.*, at most one year old). We opt for data sets that are interesting to the community and active.
(2) The benchmark should have been used in previously *published research*. Its successful use in other empirical studies proves that such a benchmark can be used for evaluating DL.
(3) The benchmark should consist of models from *different categories* (*e.g.*, recommender systems, NLP, and computer vision). We want to measure the energy consumption and run-time performance of various models.
(4) The benchmark should contain models already available for *both* PyTorch and TensorFlow. Ensuring the availability of the same models in both frameworks is necessary to compare the frameworks' energy and run-time performance.

After searching the related work (articles published in ICSE and FSE 2020-2021)[1] and GitHub, for benchmarks that can be used in the evaluation of DL, we found that DeepLearingExamples [5] satisfies our criteria listed above. DeepLearingExamples is developed by NVIDIA, a leading hardware manufacturing company.

In particular, we chose DeepLearingExamples (commit *dfed8d4*) because it is publicly available on GitHub, popular (with more than 6K stars on GitHub, and continuously maintained by NVIDIA. We preferred to rely on algorithm implementations made publicly available by practitioners to avoid introducing our own implementation biases. Furthermore, DeepLearingExamples has been used in previous research [40, 54, 61, 83], and the implementations of the DL algorithms used were developed and tested based on the original research papers proposing them. Additionally, we opted for DeepLearingExamples since this benchmark offers different models and DL algorithms, for instance, models for *Recommender Systems*, *Natural Language Processing* (NLP), and *Computer Vision*. Finally, DeepLearingExamples provides models written in both PyTorch and TensorFlow. To the best of our knowledge, DeepLearningExamples is the only repository that offers the implementations of a same DL algorithm in three different frameworks (*i.e.*, TensorFlow, PyTorch, and MXNet) for various tasks. However, we only focused on TensorFlow and PyTorch, because only one algorithm (*i.e.*, ResNet-50) was implemented in MXNet.

Models. DeepLearingExamples provides 46 state-of-the-art models that one can to train and deploy on GPUs via Docker images. From these 46 models, we selected and successfully configured six models listed in Table 1. We selected the six models using the following criteria.

(1) These models are implemented both in PyTorch and TensorFlow, while other models are not. We ensured, through manual analysis, that the models found in DeepLearningExamples, for PyTorch and TensorFlow, are identical to each other in terms of functionalities and configurations.
(2) The models can be successfully executed on our execution environment. For this, we tested each model to make sure that they can run on our available hardware.
(3) The models are unique. For instance, for computer vision, we removed different versions of the ResNet, leaving only ResNet-50 in our data set. We performed this check as we wanted to compare the energy consumption and run-time performance of different models.

Table 1 lists the models obtained according to our criteria. We have one model that belongs to the category of the recommender systems and three models that belong to the NLP category. Additionally, from the computer vision category, we kept three unique models out of five, because the other three models shared a considerable overlap.

We needed to configure the DeepLearingExamples benchmark to the requirements of our study, so that we can perform fair comparisons between PyTorch and TensorFlow [6, 8]. We present the steps we followed. First, we checked whether all the configurations of each model are alike for the two DL frameworks. Second, we executed each model with its default configurations. Third, we changed some of the default configuration (*e.g.*, number of GPUs, epochs, batches, seeds), because many times our GPU was running out of memory or running for days without giving us a result. However, in some cases, even after reducing all the available configurations and running a model for days, we did not get any result (*e.g.*, BERT [27]). Therefore, we excluded such models from our data set.

Algorithms. In the data set we used in our study, we include six DL algorithms implemented both in PyTorch and TensorFlow. In particular, the examined models use NCF, Tranformer-XL, GNMT, ResNet-50, Mask R-CNN, and SSD. We briefly describe these algorithms in the forth column of Table 1.

3.3 Evaluation Measures

For **RQ1**, we wish to measure the energy consumption and run-time performance for different models written in the two DL frameworks considered in this study. We measured both energy consumption and run-time performance for the training and inference phases of DL. To calculate the run-time performance, we used the `time` Unix tool [9]. We considered the *real* time produced by the `time` tool. To fetch the energy consumption measurements, we used the PERF [2] and `nvidia-smi` [4] tools. At its heart, PERF uses the Running Average Power Limit (RAPL [64]) framework to report the energy consumption of a running model. RAPL uses hardware

[1]We chose ICSE and FSE since these two conferences are the most prestigious in software engineering. We checked the publications from the last two years since the DL field is emerging and evolves rapidly.

Table 1: Selected models from DeepLearningExamples.

Category	Model	Data set	Description
Recommender Systems (RS)	NCF	ml-20m	It filters and provides feedback based on the NCF specifications according to He et al. [37].
Natural Language Processing (NLP)	Transformer-XL	WikiText-103	It enables capturing longer-term dependency and resolves the context fragmentation problem, built based on the study of Dai et al. [23].
	GNMT	WMT16 EN-DE	It uses Google's Neural Machine Translation System (GNMT) to translate English to German, built based on the study of Wu et al. [87].
Computer Vision (CV)	ResNet-50	Coco 2014	It is an image classification algorithm with low complexity developed according to the study of He et al. [36].
	Mask R-CNN	Coco 2017	It is a convolution-based neural network for the model of object instance segmentation, built according to the study of He et al. [35].
	SSD	Coco 2017	It detects objects in images, using a single deep neural network following the study of Liu et al. [57].

performance counters to estimate the energy consumption of the CPU cores, package (PKG), *i.e.*, core and uncore components of the processor, and main memory (RAM). We used RAPL since it is a well-established utility that has been used in related work [24, 60, 71]. Furthermore, RAPL's accuracy has been validated by various studies [26, 45, 46, 66] and it offers a high sample interval (a single reading per one millisecond). To collect the energy measurements from the GPU, we used the NVIDIA-SMI, a command-line tool developed and maintained by NVIDIA. To our knowledge, NVIDIA-SMI is the only available tool to fetch energy measurements from a GPU.

For **RQ2**, we argue about the accuracy of models that are efficient regarding energy consumption and run-time performance. Therefore, we measured the accuracy of the different models used in our study, including hit rate, perplexity, BLEU, Top-5 error rate, average precision, and precision. The evaluation measures are model-dependent. The selected measures are used by the authors who introduced the models used in our study [23, 35–37, 57, 87]. *Hit rate* aims to show the success rate of recommender systems, in suggesting the top items from a top-N list [25]; it is desired to have a high hit rate. *Perplexity* is an exponentiation of the entropy; lower values of this measure suggest more accurate models [42]. BLEU (BiLingual Evaluation Understudy) measures the difference between human and machine-translation output; a high BLEU score indicates a better model [67]. *Top-5 error rate* shows the fraction of test images for which the correct label is not among the five labels; low values of this measure are desired [48]. The last two measures are precision and average precision. Both measures are desired to be high.

For **RQ3**, we wish to investigate whether there exist functions called from the PyTorch or TensorFlow frameworks into the models under examination that consume considerable energy and impact run-time performance. To achieve this, we used a profiling tool called `cProfile` [7]. The tool produces run-time execution measurements of an application's function and library calls. Specifically, `cProfile` offers information such as the number of times each function is invoked by an application under test, the total time taken by each function, and the total time taken per call. Along with `cProfile`, we utilized `gprof2dot` [3], a command line tool to plot `dot` graphs from `cProfile`. By using `gprof2dot`, we were able to point out the exact path of the functions invoked by a model. This tool also helped us to analyze the source code of those functions that were energy hungry or run-time inefficient.

3.4 Execution Framework

To automate the evaluation of the energy consumption and run-time performance of the selected models, we implemented a framework called PrEngDL [6, 8] to support the reproducibility of our study. PrEngDL offers the following capabilities:

- installs all necessary packages and modules of our experiments through an *Ansible* script;
- sets up the test-bed configurations with the parameters used in our experiments to reproduce our results for each model;
- checks whether the corresponding GPU is compatible with the experiments' setup and ensures all the necessary dependencies are installed to get all the measurements;
- sets the power governor to the *Performance* mode to avoid reducing the performance of our experiments [18, 79];
- can execute all experiments multiple times and report the progress of the experiments;
- compiles a report on the executed models with mean values.

3.5 Experimental Settings

All experiments run on a server equipped with two 6th generation Intel(R) Xeon(R) Gold 6154 CPU running on 3.00 GHz as basic frequency totaling into 72 logical cores and 96 GB of main memory. We used Python 3.7 and `shell` scripts for our experiments, and the latest available docker containers (PyTorch and TensorFlow release 20.06-py3), available from the NVIDIA GPU Cloud Repository; it is updated on monthly basis and includes all the required dependencies needed for one to run the models.

Before running our experiments, we stopped all the unnecessary background processes (as suggested in other similar studies [19, 20, 39]) to let our system reach a *stable condition* (*i.e.*, where the energy consumption is stable). Then, we started executing models to obtain their energy consumption and run-time performance. After the end of each model's execution, we let the computer remain idle for one minute (using the `sleep` command). In this manner, we were able to avoid *tail power states* [13] and allow the system to reach a stable condition again (idle energy consumption) before

executing the next model [21]. Finally, to reduce any noise in our measurements, we executed the above steps ten times for each model. It took us ≈ 135 hours to collect the energy consumption and run-time performance measurements for all experiments.

4 RESULTS

4.1 Answer to RQ1

Our goal is to find which framework is more energy and run-time efficient for the models under examination. Table 2 and Table 3 present the energy consumption (in Joules) and run-time performance (in seconds) measured during the training and inference phases of our experiments, respectively. We use the abbreviations PKG for the core and non-core components of the processor, RAM for the main memory, and GPU for the graphic card. To compare our results we use the following equation:

$$p/p_{min} \tag{1}$$

Where p is the measurement and p_{min} is the minimum value of energy consumption or run-time performance for each model. If $x = p/p_{min}$, the model giving p_{min} is x times more energy or run-time efficient than p's implementation. Moreover, we report the mean values of the energy consumption of PKG (core and uncored components of a processor), RAM, and GPU for a particular model. The median values are also available in our replication package [8] and online repository [6]. The highlighted cells of the tables show which framework has the most energy or run-time efficient results for the corresponding model. Additionally, to assess for statistically significant differences and their magnitude, we report the results of the Wilcoxon Signed Rank Test [17] (with a confidence level of 0.05 and a Bonferroni correction for multiple hypothesis testing), and the non-parametric Vargha and Delaney's $\hat{A}_{12}$ statistic [81], respectively.[2]

Table 2 summarizes the results from the training phase of the models. We observe that TensorFlow outperforms PyTorch for training an NCF model by being 2.1x faster and 1.5x more energy efficient. By contrast, PyTorch scored better for the Transformer-XL and GNMT models compared to TensorFlow. In particular, PyTorch trained a Transformer-XL and GNMT model 1.7x and 3.1x faster compared to its counterpart. PyTorch consumed 1.5x fewer energy to train a Transformer-XL model viz-a-viz TensorFlow. Moreover, to train a GNMT model, PyTorch needed 2.8x less energy compared to TensorFlow. For training computer vision models, TensorFlow was much more energy and run-time efficient than PyTorch. Specifically, TensorFlow's run-time performance is 1.2x, 1.2x, and 2.7x more efficient for ResNet-50, Mask R-CNN, and SSD, respectively. Similarly, for ResNet-50, TensorFlow was 1.2x more energy efficient than PyTorch's implementation. Likewise, for Mask R-CNN, TensorFlow was 1.1x less energy demanding than PyTorch's implementation. For training an SSD model, TensorFlow was again 3.1x more energy efficient than PyTorch's implementation. The statistical tests (see Table 2) confirm that the results in the training phase are statistically different with a large effect size in 24 out 24 cases (100%), with TensorFlow significantly outperforming PyTorch in 16 out of 24 cases (76%).

Table 3 shows the results for the inference phase. From the collected results, we observe that TensorFlow achieves the best results for an NCF model. Particularly, TensorFlow needed 1.4x less time and 1.7x less energy for inference, compared to PyTorch. Again, similarly to the training process, PyTorch took less time and consumed less energy to train models under the category of NLP. Specifically, PyTorch took 2.4x and 3x less time to train a Transformer-XL and GNMT model, while it also used 2.1x and 2.7x less energy to train a Transformer-XL and GNMT model, respectively. For the computer vision category, TensorFlow only performed better than PyTorch for a ResNet-50 model. In particular, TensorFlow's implementation was 1.2x and 1.7x more run-time and energy efficient than PyTorch's, respectively. To infer a Mask R-CNN and SSD model, PyTorch was 1.3x and 1.7x faster, while also being 1.1x and 1.5x more energy efficient than TensorFlow's implementations, correspondingly. The statistical tests confirm that the results achieved by the two frameworks in the inference phase statistically differ, with a large effect size for 21 out of 24 cases (87%), and are in favor of PyTorch in 16 out of these 21 cases (76%).

> **Answer to RQ1:** We find that TensorFlow is the least costly framework for training recommender systems and computer vision models, while PyTorch is the cheapest for training NLP models. When it comes to model inference, TensorFlow is better for the recommender systems and for the ResNet-50 computer vision model, while PyTorch outperforms TensorFlow for the remaining models. Overall, we observe that the cheapest framework for the training phase is TensorFlow, while PyTorch achieves the least costly inference.

4.2 Answer to RQ2

Our goal is to find the energy and run-time performance trade-offs against the selected models' accuracy. Figure 1 depicts the accuracy of each model selected in our study, along with the corresponding run-time performance and energy consumption. To identify the run-time, energy (combined energy of PKG, RAM, and GPU), and accuracy trade-offs, we utilize the combined results of training and inference summarized in Table 2 and Table 3, along with the accuracy results illustrated in Figure 1.

We use the combined results of training and inference, because obtaining the final accuracy involves both the training and inference steps. Additionally, we compare and discuss our results by using the equation 1, as we did for RQ1. We use the basic configurations for executing the experiments, since we are interested in measuring energy and run-time performance, and we do not tune the configurations such that to achieve the best accuracy. To gather accuracy results, we run only once our techniques since according to the related work [74], using the same and basic configurations, the models will be deterministic, and accuracy will remain the same.

Figure 1 presents the trade-offs among the three aspects *i.e.*, energy consumption (y-axis), run-time performance (size of circle), and accuracy (color of the circle). For example, TensorFlow's

[2] Given a performance measure M, the $\hat{A}_{12}$ statistic measures the probability, where implementing a model with a framework A yields better results than implementing it with a framework B. If the two implementations are equivalent, we will get $\hat{A}_{12} = 0.5$. If the first implementation performs better than the second one, we can have the following cases: $\hat{A}_{12}$ is considered small for $0.6 \leq \hat{A}_{12} < 0.7$, medium for $0.7 < \hat{A}_{12} < 0.8$, and large for $\hat{A}_{12} \geq 0.8$.

Table 2: RQ1. Mean values for the energy consumption (in Joules) and run-time performance (in seconds) of training. The Wilcoxon statistical significance test results (p-value) and effect size ($\hat{A}_{12}$) are also reported.

Model	PKG Energy			RAM Energy			GPU Energy			Run-Time		
	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)
NCF	327,561	**280,428**	< 0.001(1)	51,537	**29,300**	< 0.001(1)	173,505	**85,127**	0.001(0.9)	5,901	**2,710**	< 0.001(1)
Transformer-XL	**144,781**	201,979	< 0.001(1)	**15,151**	24,791	< 0.001(1)	**113,754**	187,333	< 0.001(1)	**1,431**	2,495	< 0.001(1)
GNMT	**195,975**	604,856	< 0.001(1)	**25,164**	75,908	< 0.001(1)	**196,911**	521,321	< 0.001(1)	**2,515**	7,805	< 0.001(1)
ResNet-50	368,245	**258,787**	< 0.001(1)	39,506	**31,752**	< 0.001(1)	236,041	**216,157**	< 0.001(1)	3,472	**2,826**	< 0.001(1)
Mask R-CNN	255,395	**198,027**	< 0.001(1)	27,897	**22,739**	< 0.001(1)	176,012	**146,886**	< 0.001(1)	2,438	**2,028**	< 0.001(1)
SSD	590,050	**185,356**	< 0.001(1)	63,835	**22,678**	< 0.001(1)	412,664	**120,646**	< 0.001(1)	5,667	**2,072**	< 0.001(1)

Table 3: RQ1. Mean values for the energy consumption (in Joules) and run-time performance (in seconds) of inference. The Wilcoxon statistical significance test results (p-value) and effect size ($\hat{A}_{12}$) are also reported.

Model	PKG Energy			RAM Energy			GPU Energy			Run-Time		
	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)	PyTorch	TensorFlow	p-value ($\hat{A}_{12}$)
NCF	16,697	**11,275**	0.48(0.6)	1,780	**1,231**	0.48(0.6)	5,049	**1,829**	< 0.001(1)	159	**113**	0.48(0.6)
Transformer-XL	**21,789**	42,037	< 0.001(1)	**2,333**	4,657	< 0.001(1)	**14,968**	38,769	< 0.001(1)	**208**	509	< 0.001(1)
GNMT	**3,205**	9,638	< 0.001(1)	**346**	992	< 0.001(1)	**1,465**	3,406	< 0.001(1)	**31**	96	< 0.001(1)
ResNet-50	88,653	**36,531**	< 0.001(1)	9,333	**5,356**	< 0.001(1)	53,168	**45,812**	< 0.001(1)	776	**603**	< 0.001(1)
Mask R-CNN	**103,321**	106,301	< 0.001(0.9)	**11,120**	13,529	< 0.001(1)	**70,858**	81,806	< 0.001(1)	**963**	1,337	< 0.001(1)
SSD	**42,906**	67,035	< 0.001(1)	**4,554**	8,048	< 0.001(1)	**17,122**	26,893	< 0.001(1)	**408**	717	< 0.001(1)

implementation of GNMT not only consumes higher energy than PyTorch's implementation, but also takes longer to execute (and, thus, it has a larger circle compared to PyTorch). However, TensorFlow's GNMT shows slightly better accuracy than the PyTorch's implementation. Therefore, the circles have the same color.

For the NCF model, TensorFlow not only outperforms PyTorch in terms of energy consumption and run-time performance, but also in terms of accuracy, having 1.1x better hit rate. For the Transformer-XL model, we observe that PyTorch is 1.2x more accurate. However, for the GNMT model, PyTorch is 1x more inaccurate than TensorFlow. For the computer vision models, we find that TensorFlow has better accuracy for the ResNet-50 model, since the top-5 error rate of PyTorch is 44x higher than TensorFlow's. We also observe for the SSD model that TensorFlow has 2x better precision compared to PyTorch. For Mask R-CNN, we obtain zero results for both frameworks, as the used configurations and hyper-parameters did not contribute to a visible accuracy.

Answer to RQ2: The collected results suggest that better energy consumption and run-time performance—in most cases—yield better accuracy results as well. Overall, we find that TensorFlow has similar accuracy to PyTorch, under the configurations and parameters used in our study.

Table 4: RQ3. Energy and run-time performance tests.

Tuples	PyTorch	TensorFlow
PKG Energy–Run-Time	0.25	0.88
RAM Energy–Run-Time	0.88	0.94
GPU Energy–Run-Time	0.42	0.60

4.3 Answer to RQ3

Our goal is to identify APIs of the frameworks under examination that consume a significant amount of energy and are run-time inefficient. We analyze such functions, which the models invoke, that contribute more than 1% to the total execution time of the training and inference of a model (see tables 5, 6, 7, and 8). We select such a low threshold since any function call with lower than 1% of execution time might have a negligible impact on the energy and run-time performance of the training and inference of a model. The complete profiling of our experiments is available in our replication package [8] and online repository [6]. Table 5, Table 6, Table 7, and Table 8 present the collected results for each model, the corresponding function name, number of calls, the total execution time (run-time), and the portion, in terms of percentage, of the total execution time.

To the best of our knowledge, there are no available tools to measure the energy consumption of a model in terms of milliseconds; thus, it is challenging for one to collect energy measurements with a high sampling rate and map them to the energy consumption of the selected models. Additionally, since the total energy consumption is a function of the power usage multiplied by the total time spent on a model (see Section 2), we consider that greater total time taken by a function (of PyTorch or TensorFlow) also increases energy consumption. Therefore, in Table 5, Table 6, Table 7, and Table 8, we report only the run time. Furthermore, we perform the Spearman's correlation test (since our data does not follow a normal distribution), and find that PyTorch and TensorFlow's energy consumption and run-time performance have, on average, a positive moderate and very strong monotonic correlation, respectively (see Table 4). Table 4 presents the results for the correlation test of each framework and the resources tested. The detailed call-graphs used, along with the performance measures (in figures), can be accessed in our replication package [8] and online repository [6].

Table 5: RQ3. PyTorch training. Function name, number of calls (Ncalls), run-time, cost against the total run-time (%), and type where ■ stands for complex calculations, □ complex implementation, • large data, ◇ device dependency, and – unknown. The same symbols apply to Tables 6, 7, 8.

Model	Function Name	Ncalls	Run-Time	Cost	Type
NCF	Torch.autograd.backward	1,551,700	2,348	39%	–
	Torch.addmm	6,206,856	374	6.2%	■
	Torch.cat	3,103,434	308	5.1%	□
	Torch.nn.functional.embedding	6,206,856	234	3.8%	•
	Torch.nn.functional.dropout	4,655,142	171	2.8%	•
	Torch.nn.functional.relu	4,655,142	131	2.1%	■
	Torch.Tensor.t	6,206,856	66	1%	–
Transformer-XL	Torch.autograd.backward	4,000	950	48.9%	–
	Torch.Tensor.item	79,160	376	19.3%	–
	Torch.Tensor.nonzero	18,628	142	7.3%	–
GNMT	Torch.autograd.backward	27,327	1,218	47.4%	–
	Torch.Tensor.sum	54,823	67	2.6%	■
	Torch.Tensor.mul_	3,229,116	51	1.9%	■
	Torch.Tensor.add_	2,389,992	37	1.4%	■
ResNet50	Torch.cuda.synchronize	7,404	2,880	83.6%	◇
	Torch.autograd.backward	7,404	331	9.6%	–
	Torch.nn.Conv2d	392,412	61	1.9%	■
	Torch.Tensor.item	7,524	44	1.2%	–
Mask R-CNN	Torch.Tensor.item	37,960	1,013	52.9%	–
	Torch.tensor	48,524	515	26.9%	◇
	Torch.autograd.backward	1,000	54	2.8%	–
SSD	Torch.Tensor.zero_	2,706,186	1,143	21.9%	□
	Torch.cuda.synchronize	14,786	1,051	20.2%	◇
	Torch.Tensor.add_	5,411,493	522	10%	■
	Torch.autograd.backward	14,786	329	6.3%	–
	Torch.Tensor.mul_	2,705,655	174	3.3%	■

PyTorch Training. The obtained results for PyTorch indicate that the functions of APIs such as `Torch.autograd.backward` and `Torch.cuda.synchronize` contribute to the most time taken in training a model (see Table 5). Specifically, we find that all of our models spend on average 25.6% of their total execution time on the function `run_backward`, which is invoked by the `Torch.autograd.backward` API, responsible for computing a tensor's gradients. We also observe that the `Torch.cuda.synchronize` API can take up to 51.9%, on average, for training an ResNet-50 or SSD model. The corresponding function waits for all kernels of all streams for the GPU card to complete. Apart from the aforementioned functions, we find that a function called by the `Torch.Tensor` object—such as `t, item, nonzero, sum, zero_, add_`, and `mul_`—takes up, on average, to 11.1% of the total execution time. Functions to the object `Torch.Tensor`, such as `add_, mul_`, and `sum`, are responsible for performing arithmetic calculations on tensors, while `item` returns the value of a tensor, `t` is responsible to transpose dimensions, and so on. Finally, for training a recommender system, we observe that functions such as `Torch.addmm` (matrices multiplication), `Torch.cat` (tensors concatenation), `Torch.nn.functional.embedding` (retrieval of word embeddings using indices), and `Torch.nn.functional.relu` (applies a rectified linear function to given elements) contribute to 6.2%, 5.1%, 3.8%, and 2.1% of the total execution time, respectively.

PyTorch Inference. In contrast with PyTorch's training process, for inference, there is not a single API, for all our models, that is responsible for taking the most of the execution time. However, as in the training process, in ResNet-50, `Torch.cuda.synchronize` and `Torch.nn.Conv2d` functions take up to the most of the inference time. For NCF, Transformer-XL, ResNet-50, and Mask R-CNN, we observe that similar functions are invoked in the training process—except for the `Torch.autograd.backward` being invoked in the training only—and contribute to the most of the execution time.

TensorFlow Training. When it comes to TensorFlow's training process, we find that all the models, apart form Mask R-CNN, use the `TensorFlow.Session.run` API at 90.3%, on average, of their total training time. The corresponding API is responsible for running computations and evaluating the tensors. The specific function that is eventually invoked by `TensorFlow.Session.run` is the `_pywrap_tensorflow_internal.TF_SessionRun_wrapper`. For Mask R-CNN, we observe that `TensorFlow.keras.Model.fit` takes 98.7% of the total execution time to train the model.

TensorFlow Inference. `TensorFlow.Session.run` is found to take the most of the execution time for model inference, which is 55.6%, on average. In comparison with the whole model implementation, we observe that NCF's inference process is not affected to a great extend by the TensorFlow framework. This possibly happens because many other libraries are used by the developers in

Table 6: RQ3. PyTorch inference. Functions, number of calls (Ncalls), run-time, cost against the total run-time (%), and type.

Model	Function Name	Ncalls	Run-Time	Cost	Type
NCF	Torch.addmm	874,240	17	19.1%	■
	Torch.cat	437,123	15	16.6%	□
	Torch.nn.functional.embedding	874,240	8	9.4%	•
	Torch.nn.functional.relu	655,680	5	6.2%	■
	Torch.Tensor.t	874,240	2	2.8%	–
	Torch.sigmoid	218,560	2	2.3%	–
Transformer-XL	Torch.Tensor.item	87,945	69	32.6%	–
	Torch.Tensor.nonzero	19,544	38	18%	–
	Torch.einsum	234,528	15	7.1%	■
	Torch.cat	175,896	6	2.9%	□
	Torch.Tensor.matmul	390,880	6	2.8%	■
GNMT	Torch.Tensor.to	117	1	3.6%	□
	Torch.nn.lstm	7680	1	2.9%	■
	Torch.Tensor.matmul	7,584	1	1.3%	□
ResNet50	Torch.cuda.synchronize	4,650	644	84.5%	◇
	Torch.nn.Conv2d	246,450	24	3.2%	■
Mask R-CNN	Torch.tensor	81,819	578	60.6%	◇
	Torch.nn.Conv2d	380,000	24	2.6%	■
	Torch.Tensor.nonzero	400,000	17	1.8%	–
	Torch.Tensor.float	1,505,005	16	1.7%	•
	Torch.Tensor.to	105,312	16	1.7%	□
	Torch.Tensor.type	9,735,395	14	1.5%	–
SSD	Torch.max	455,439	6	1.3%	•
	Torch.nn.Conv2d	10,075	6	1.3%	■
	Torch.Tensor.item	465,344	5	1.3%	–

Table 7: RQ3. TensorFlow training. Functions, number of calls (Ncalls), run-time, cost against the total run-time (%), and type.

Model	Function Name	Ncalls	Run-Time	Cost	Type
NCF	TensorFlow.Session.run	1,551,702	2,325	69.3%	□
Transformer-XL	TensorFlow.Session.run	4,004	1,447	96.4%	□
GNMT	TensorFlow.Session.run	30,718	7,654	95.4%	□
ResNet50	TensorFlow.Session.run	7,509	2,825	97.8%	□
Mask R-CNN	TensorFlow.keras.Model.fit	1,004	1,993	98.7%	•
SSD	TensorFlow.Session.run	20,101	1,920	92.7%	□

Table 8: RQ3. TensorFlow inference. Functions, call number (Ncalls), run-time, cost against the total run-time (%), and type.

Model	Function Name	Ncalls	Run-Time	Cost	Type
NCF	TensorFlow.Session.run	34	4	1.2%	□
Transformer-XL	TensorFlow.Session.run	1,630	481	96.4%	□
GNMT	TensorFlow.Session.run	30	43	31.7%	□
	TensorFlow.io.gfile.GFile	8,142,390	16	12.1%	□
ResNet50	TensorFlow.Session.run	5,006	575	97.6%	□
Mask R-CNN	TensorFlow.Keras.Model.predict	1,559	830	84.6%	•
SSD	TensorFlow.Session.run	5,007	371	52.5%	□
	TensorFlow.io.gfile.Glob	1	298	42.2%	□

the inference phase, apart from TensorFlow, taking up to the most of the total execution time. Moreover, for Mask R-CNN, we observe that `TensorFlow.keras.Model.predict` consumes 84.6% of the total execution time in inference. We also observe two classes from `TensroFlow.io.gfile`, namely `GFile` (an asynchronous file I/O wrapper) and `Glob` (returns a list of files by using pattern matching).

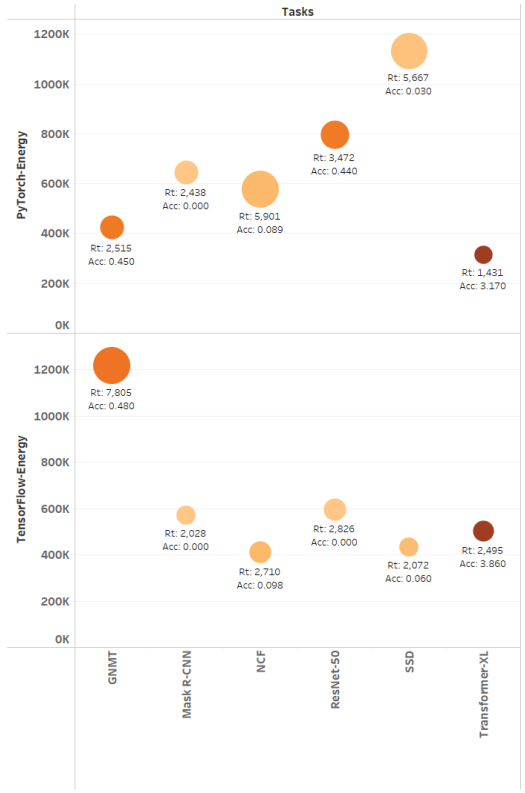

Figure 1: RQ2. Comparison of energy-consumption, run-time performance (Rt), and accuracy (Acc) between TensorFlow and PyTorch. The size of a circle represents the run-time performance (*i.e.*, the bigger the circle, the lower the run-time) and color represents the value of accuracy metric (*i.e.*, the darker the circle, the higher the accuracy); the interpretation depends on the metric of the model. Accuracy has been computed with Hit Rate (NFC), Perplexity (Transformer-XL), BLEU (GNMT), Top-5 error rate (ResNet-50), Avg. Precision (Mask R-CNN) and Precision (SSD).

Although `Glob` is invoked only a single time, it takes a huge toll on the inference process of SSD (*i.e.*, 42.2% of the total execution time). Likewise, `GFile` calls take 12.1% of the total execution time in the inference of the GNMT model.

Answer to RQ3: Most of the times, specific APIs are most demanding and take the majority of the training and inference time of the models examined. Overall, PyTorch's `autograd.backward` and TensorFlow's `Session.run` are the most performance intensive API calls.

4.4 Discussion

Analysis of Documentation and Source Code. Our results show that there are functions in the APIs of PyTorch and TensorFlow that may slow the execution of DL applications and consume more energy. Therefore, the first two authors of this paper manually investigated the functions' documentation and source code (function body) in the PyTorch and TensorFlow frameworks for all the functions listed in Tables 5, 6, 7, and 8. In case of a disagreement between the two authors, there was a discussion, between them, to reach a consensus [6, 8].

Our manual analysis revealed that in the documentation of these functions there is not any particular information regarding energy consumption or run-time performance. We also read the source code of the functions identified and found four types of issues that could be improved in the future. As it is shown in Tables 5, 6, 7, and 8, we identified: (1) nine unique functions performing complex calculations, (2) six unique functions having complex implementation (*e.g.*, with many control flows), (3) seven unique functions that can handle large data, and which performance depends on the data magnitude, and (4) two unique functions which performance depends on the device characteristics used. We also found six methods in PyTorch that we were unable to categorize, because these methods were either written in another language than Python or because we lack the knowledge to understand their source code.

Suggestions. Based on our manual analysis, we suggest that since there is a need for *greener* DL frameworks [77], extra information on energy consumption, run-time performance, and minimum required configurations should be systematically included in the documentation of such frameworks, in the future.

DL frameworks could, for instance, include in their documentation new "energy and performance-related text fields" for each method and special symbols for each method type (*e.g.*, as we do in the last column of Table 5) to indicate the requirements. For example, the PyTorch documentation [1] is currently limited, and in the future it would be good to include the minimum hardware requirements, at least.

Developers could also list constraints regarding energy and performance requirements, possible errors, or energy and performance issues that can happen at run-time, as well as provide default values to avoid run-time issues. Furthermore, annotations (*e.g.*, as `@NonNull` in Java 8) could be added to the documentation to warn developers about potential energy/performance inefficiencies, which could be encountered at run-time. These observations could be drawn from the testing and evaluation of the energy and performance of DL frameworks. Using PrEngDL, we hope that energy and performance metrics of DL frameworks could be gathered systematically. We, however, acknowledge that these suggestions should be further validated in future work, possibly involving researchers and practitioners.

For functions that are complex or handle large data, DL researchers may consider to optimize the performance of existing algorithms themselves, and, then, DL frameworks can be updated accordingly. For functions with a complex implementation, the developers of DL frameworks may consider directly optimize their existing code by leveraging, for example, the Genetic Improvement of software [72] (which studies how to automatically modify the source code to improve software's non-functional properties).

In the following paragraphs, we summarize the *key take-aways* of our study for researchers and practitioners that develop DL frameworks. Furthermore, we suggest *steps* that a user of a DL framework could follow, according to the insights of our study, to write more efficient programs when using PyTorch and/or TensorFlow.

Key Take-aways. To summarize, our study offers the following take-aways to researchers and practitioners that use DL frameworks:

- The study raises awareness on the fact that different DL frameworks incur different energy and performance costs for both the training and inference phases. Our results show that no framework is best for all tasks investigated and, thus, call for actions to allow users to consider and understand energy and performance requirements when selecting a DL framework.
- The proposed approach (PrEngDL) can help future studies in this domain, by offering a comprehensive and sound method to automatically measure and compare energy and run-time performance.
- The deficiencies we found, in the current documentation of DL frameworks, reveal the need for new *green-aware documentation* of DL frameworks.

Steps. To improve efficiency, a user of a DL framework may consider the following steps: (1) run small experiments first, and test configurations, before running large-scale experiments, (2) use profiling approaches on small experiments to estimate resources to be used for large experiments afterwards, (3) analyze the energy-accuracy trade-offs to decide whether energy and performance should be sacrificed over accuracy.

To conclude, we hope that the methodology presented in this paper, together with its publicly available implementation (PrEngDL [6, 8]), can aid users to assess DL frameworks' energy and run-time performance.

5 THREATS TO VALIDITY

Internal Validity. Due to hardware constraints, we used basic configurations and parameters of the models to facilitate the training process on our test-bed. The reader should consider that some of the obtained results may vary significantly based on the used GPUs or CPU architectures. Using different configurations can impact the accuracy score, reported in Figure 1. However, our current aim is to investigate the energy and run-time performance of two DL frameworks while performing the same models. Therefore, we plan to examine the impact of different configurations on accuracy in the future. Also, in Section 4.3, we found that two functions under the `gfile` module of PyTorch have a heavy toll on the performance of the framework. This fact could have been caused by our server system's slow hard-disk drive. However, for the fair comparisons of our findings, we calculated correlation scores, in Table 4. We acknowledge that hardware is *not* always a limitation, for example for large companies accessing large computing resources. Thus, we wish to run our experiments in an industrial setting, in the future, to show the implications of hardware on our results.

The energy and the run-time performance of an application under test can be affected by many different factors, such as running background processes, daemons, and so on. We tried to limit as much as possible such interference. Having full control over the OS' workload and background operations is hard, because, at any time, different daemons, for instance, may operate. This could affect our calculations, too, and the results may vary among different executions. However, this issue is common in such studies. We also set one minute of idle time between each execution of our tasks, because we found that this time budget is sufficient for our system to reach a stable condition. Using another time budget may give different results.

Finally, our manual analysis, for the functions found in Section 4.3, many suffer from human errors. To eliminated this issue, two validators cross-checked the results and we made our results publicly available [6, 8].

External Validity. Regarding the generalizability of our findings (Section 4), we admit that our results regard the models and frameworks selected in our study. However, we argue, that the benchmarks used are developed from well-known research studies as mentioned in Section 3. In the future, we wish to execute our study using other benchmarks to strengthen the generalizability of our findings and balance the categories of the models used in our experiments (Table 1). We note that we kept the same version of the packages and modules used originally in the DeepLearningExamples repository, since these tasks are extensively tested and developed on monthly basis according to NVIDIA's developers [5]. Therefore, under different versions the results may differ slightly.

Reliability. For the reproducibility of our study, we developed an execution framework, PrEngDL, and we made it publicly available, as well as the inputs and outputs of our experiments [6, 8]. We also provide a configuration management script to enable the installation of modules and tools needed to run our tasks.

6 RELATED WORK

Many studies examine the accuracy of DL. However, a few studies focus on DL's energy consumption and run-time performance [77]. Researchers have pointed out though that the field of DL is energy demanding. Thus, DL also contributes to the increased CO_2 emissions, and has a great financial cost as well [14, 15, 55, 76, 78, 84].

Several research studies introduced practices on how to use traditional ML efficiently to reduce energy consumption. For instance, McIntosh et al. [59] performed an empirical study to examine which algorithms are less energy demanding to train ML models for Android devices. Their results suggest that J48, SMO, and MLP contribute to more energy efficiency, better accuracy, and show a correlation to algorithms' complexity. Moreover, the authors pointed out a number of factors that can significantly affect the energy consumption of ML for Android devices, such as the data set size and the number of fields. To suggest changes in Java-based source code for ML, Kumar et al. [49] introduced JEPO, an Eclipse plugin that can help in optimizing ML source code regarding data types, operators, strings, exceptions, objects, and arrays. In addition to the previously mentioned studies, Kan et al. [44] proposed the eclass that makes use of the Dynamic Voltage Frequency Scaling mechanisms to reduce the energy consumption of a computer, while training a ML model. The suggested approach increased the average energy savings by 9.1%.

Furthermore, researchers performed studies to investigate and suggest changes for popular DL models to make them more efficient. For instance, Zhang et al. [89] conducted a preliminary study to find the latency, memory footprint, and energy usage of AlexNet and SqueezeNet implemented using TensorFlow, TensorFlow Lite, PyTorch, MXNet, and Caffe2. As a result, the authors found that there is not a single framework that outperforms the others.

Wang et al. [84] proposed an approach of dropping unnecessary computations from CNN models running on FPGAs to reduce energy consumption. The authors achieved energy savings ranging from 60% to 90%, with an accuracy loss of 1.2% to 2% for ResNet74, ResNet110, and MobileNetV2, respectively.

Other researchers proposed guidelines on how to fetch energy measurements for studies with respect to ML [32]. Specifically, the study of García-Martín et al. [32] states the limitations of various approaches used by researchers to estimate the energy consumption of model training and proposes ways to build models that can estimate the energy consumption on different hardware systems. Likewise, in the study of Fu et al. [31], the authors developed models to estimate the energy consumption of machine learning applications. Similarly, Pathak et al. [70] used system calls to generate power models for estimating smart-phones' energy consumption. Furthermore, Bornholt et al. [13] argued that using only a ML model is not enough to estimate the energy consumption of applications.

This study. We examined how two of the most popular DL frameworks (PyTorch and TensorFlow) perform for DL models from different categories. Closer to our study is the preliminary work by Zhang et al. [89]. Zhang et al. [89] assessed the performance of running a trained model (*i.e.*, not during the training phase) with the purpose of assessing the use of different hardware for two pre-trained models, namely AlexNet and SqueezeNet, from one category (*i.e.*, image classification). By contrast, we measured the performance during both the training and inference phases of six different models from three different categories (*i.e.*, recommender system, NLP, and computer vision).[3] Besides, our empirical methodology is more robust including, for instance, statistical tests, effect size, and mitigations for stochastic behavior. Additionally, we found which DL framework is more energy and run-time efficient for certain models and investigated the trade-offs of the frameworks accuracy. Finally, we investigated the reasons behind the obtained results concerning which functions or libraries are hurting the performance of the frameworks examined, and we provided some initial suggestions for DL frameworks' users and makers/developers.

7 CONCLUSIONS

High accuracy of DL comes at the cost of high computational cost and resource utilization. In this study, we studied and compared the energy consumption and run-time performance of two commonly used DL frameworks (PyTorch and TensorFlow). We found that TensorFlow performs better for training models of the recommender systems and computer vision categories, and PyTorch of the NLP category. Regarding inference, TensorFlow performs better than PyTorch only for the recommender systems and ResNet-50 categories. Furthermore, we found that better energy consumption and run-time performance—in most cases—yield better accuracy results. Overall, TensorFlow is more energy and run-time efficient compared to PyTorch in the training phase, but less efficient for the inference phase. We also identified specific APIs and functions (from PyTorch and TensorFlow) that are most resource-hungry and take the majority of the training and inference time.

Our results can have several **implications** for researchers and practitioners:

- DL frameworks show a significant model-sensitive difference in their run-time performance and energy consumption. Therefore, DL developers may choose the most appropriate framework for the model at hand, while keeping accuracy, run-time performance, and energy consumption optimal.
- The training phase of DL frameworks is more expensive than the inference one, thus, resulting in a higher footprint impact. Consequently, DL users should consider appropriate steps when using DL models with large data.
- Our manual analysis of the source code and documentation of DL frameworks reveals that the current documentation needs improvement [22], since it lacks, for example, information about the minimum hardware requirements regarding energy consumption and run-time performance. Therefore, users are left with no indication on how to choose a DL framework with regards to these aspects. These results raise the awareness of the need for *greener* software for users, DL library makers/developers, and researchers.
- Our manual code analysis and profiling identifies the most expensive APIs. These findings can guide both researchers and DL library makers/developers into the optimization of the energy and performance of DL source code, which could be attempted both manually or automatically by using, for instance, the Genetic Improvement of software [72].

Acknowledgements. Maria Kechagia and Federica Sarro are supported by the ERC grant no. 741278 (EPIC).

[3] Our choice of models is guided by maximizing the number of different categories and algorithms studied, while using publicly available, actively maintained, and tested implementations, in two DL frameworks. The models (AlexNet and SqueezeNet) used by Zhang et al. [89] cannot be used in our study, because SqueezeNet is not implemented for TensorFlow and AlexNet provides only a pre-trained model. For our study, we need access to the source code to train the models and take measurement during this phase. However, we could not locate the training source code for AlexNet and SqueezeNet.

REFERENCES

[1] [n.d.]. https://pytorch.org/docs/stable/generated/torch.cuda.synchronize.html?highlight=torch%20cuda%20synchronize#torch.cuda.synchronize
[2] 2020. *perf: Linux profiling with performance counters.* https://perf.wiki.kernel.org/index.php/Main_Page
[3] 2021. *GProf to dot.* https://github.com/jrfonseca/gprof2dot
[4] 2021. *NVIDIA System Management Interface.* https://developer.nvidia.com/nvidia-system-management-interface
[5] 2021. *NVIDIA/DeepLearningExamples.* Retrieved 2021-09-01 from https://github.com/NVIDIA/DeepLearningExamples
[6] 2021. *Online repository for the paper Green AI: Do Deep Learning Frameworks Have Different Costs?* https://github.com/stefanos1316/ICSE_2022_artifact
[7] 2021. *The Python Profilers.* https://docs.python.org/3/library/profile.html
[8] 2021. *Replication package for the paper Green AI: Do Deep Learning Frameworks Have Different Costs?* https://doi.org/10.5281/zenodo.6029576
[9] 2021. *time(1) - Linux man page.* https://linux.die.net/man/1/time
[10] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. 265–283.
[11] Carol V Alexandru, Sebastiano Panichella, and Harald C Gall. 2017. Replicating parser behavior using neural machine translation. In *Proceedings of the 25th International Conference on Program Comprehension.* IEEE Press, 316–319.
[12] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 334–344. https://doi.org/10.1109/ICSME.2016.31
[13] J. Bornholt, T. Mytkowicz, and K. S. McKinley. 2012. The model is not enough: Understanding energy consumption in mobile devices. Hot Chips, 1–3.

[14] A. Canziani, Adam Paszke, and E. Culurciello. 2016. An Analysis of Deep Neural Network Models for Practical Applications. *ArXiv* abs/1605.07678 (2016).

[15] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 750–762. https://doi.org/10.1145/3368089.3409759

[16] Shaiful Alam Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei. 2019. GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1107–1118. https://doi.org/10.1109/ICSE.2019.00114 ISSN: 1558-1225.

[17] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2 ed.). Routledge, New York. https://doi.org/10.4324/9780203771587

[18] Ivan Tomas Cotes-Ruiz, Rocio P. Prado, Sebastian Garcia-Galan, Jose Enrique Munoz-Exposito, and Nicolas Ruiz-Reyes. 2017. Dynamic Voltage Frequency Scaling Simulator for Real Workflows Energy-Aware Management in Green Cloud Computing. *PLOS ONE* 12, 1 (Jan. 2017), e0169803. https://doi.org/10.1371/journal.pone.0169803

[19] Luís Cruz. 2021. 16 Guidelines for Effective Data Visualizations in Academic Papers. http://luiscruz.github.io/2021/03/01/effective-visualizations.html. Blog post.

[20] Luís Cruz. 2021. Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments. http://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post.

[21] 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.

[22] Alex Cummaudo, Rajesh Vasa, John Grundy, and Mohamed Abdelrazek. 2020. Requirements of API Documentation: A Case Study into Computer Vision Services. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/tse.2020.3047088

[23] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 2978–2988. https://doi.org/10.18653/v1/P19-1285

[24] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. 2010. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. 189–194. https://doi.org/10.1145/1840845.1840883

[25] Mukund Deshpande and George Karypis. 2004. Item-based top-N recommendation algorithms. *ACM Transactions on Information Systems* 22, 1 (Jan. 2004), 143–177. https://doi.org/10.1145/963770.963776

[26] 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* (Alexandria, VA, USA) *(MEMSYS '16)*. Association for Computing Machinery, New York, NY, USA, 455–470. https://doi.org/10.1145/2989081.2989088

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[28] Michael D Ernst. 2017. Natural language is a programming language: Applying natural language processing to software development. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[29] Daniel J Felleman and David C Van Essen. 1991. Distributed Hierarchical Processing in the Primate Cerebral Cortex. *Cerebral Cortex* 1, 1 (1991), 1–47.

[30] Stephen R Foster, William G Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 222–232.

[31] Cuijiao Fu, Depei Qian, and Zhongzhi Luan. 2018. Estimating Software Energy Consumption with Machine Learning Approach by Software Performance Feature. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 490–496. https://doi.org/10.1109/Cybermatics_2018.2018.00106

[32] 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. https://doi.org/10.1016/j.jpdc.2019.07.007

[33] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[34] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI*. 1345–1351.

[35] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2020. Mask R-CNN. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 2 (2020), 386–397. https://doi.org/10.1109/TPAMI.2018.2844175

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[37] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 173–182. https://doi.org/10.1145/3038912.3052569

[38] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[39] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 12–21.

[40] Sitao Huang, Carl Pearson, Rakesh Nagi, Jinjun Xiong, Deming Chen, and Wenmei Hwu. 2019. Accelerating Sparse Deep Neural Networks on FPGAs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2019.8916419

[41] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

[42] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. 1977. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America* 62, S1 (Dec. 1977), S63–S63. https://doi.org/10.1121/1.2016299 Publisher: Acoustical Society of America.

[43] Rie Johnson and Tong Zhang. 2015. Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 103–112.

[44] Edward Y. Y. Kan, W. K. Chan, and T. H. Tse. 2012. EClass: An execution classification approach to improving the energy-efficiency of software via machine learning. *Journal of Systems and Software* 85, 4 (2012), 960–973. https://doi.org/10.1016/j.jss.2011.11.1010

[45] Richard Kavanagh and Karim Djemame. 2019. Rapid and accurate energy models through calibration with IPMI and RAPL. *Concurrency and Computation: Practice and Experience* 31, 13 (2019), e5124. https://doi.org/10.1002/cpe.5124

[46] 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 (March 2018), 26 pages. https://doi.org/10.1145/3177754

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[48] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[49] Mohit Kumar, Xingzhou Zhang, Liangkai Liu, Yifan Wang, and Weisong Shi. 2020. Energy-Efficient Machine Learning on the Edges. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 912–921. https://doi.org/10.1109/IPDPSW50202.2020.00153

[50] Varsha S. Lalapura, J. Amudha, and Hariramn Selvamuruga Satheesh. 2021. Recurrent Neural Networks for Edge Intelligence: A Survey. *ACM Comput. Surv.* 54, 4, Article 91 (May 2021), 38 pages. https://doi.org/10.1145/3448974

[51] James Laros III, Kevin Pedretti, Suzanne M. Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. 2013. Energy Delay Product. In *Energy-Efficient High Performance Computing*. Springer London, 51–55. https://doi.org/10.1007/978-1-4471-4492-2_8

[52] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. 1997. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks* 8, 1 (1997), 98–113.

[53] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[54] Bingbing Li, Zhenglun Kong, Tianyun Zhang, Ji Li, Zhengang Li, Hang Liu, and Caiwen Ding. 2020. Efficient Transformer-based Large Scale Language Representations using Hardware-friendly Block Structured Pruning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 3187–3199. https://doi.org/10.18653/v1/2020.findings-emnlp.286

[55] Da Li, Xinbo Chen, Michela Becchi, and Ziliang Zong. 2016. Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing*

and Communications (SustainCom) (BDCloud-SocialCom-SustainCom). 477–484. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.76

[56] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 599–609.

[57] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 21–37.

[58] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An Empirical Study of Practitioners' Perspectives on Green Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2884781.2884810

[59] Andrea McIntosh, Safwat Hassan, and Abram Hindle. 2018. What can Android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering* (10 May 2018), 1–42.

[60] I. Moura, G. Pinto, F. Ebert, and F. Castor. 2015. Mining Energy-Aware Commits. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 56–67. https://doi.org/10.1109/MSR.2015.13

[61] Takuya Narihira, Javier Alonsogarcia, Fabien Cardinaux, Akio Hayakawa, Masato Ishii, Kazunori Iwaki, Thomas Kemp, Yoshiyuki Kobayashi, Lukas Mauch, Akira Nakamura, Yukio Obuchi, Andrew Shin, Kenji Suzuki, Stephen Tiedmann, Stefan Uhlich, Takuya Yashima, and Kazuki Yoshiyama. 2021. Neural Network Libraries: A Deep Learning Framework Designed from Engineers' Perspectives. arXiv:2102.06725 [cs.LG]

[62] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2013. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 651–654.

[63] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 574–584.

[64] Srinivas Pandruvada. 2014. *NVIDIA System Management Interface*. https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl

[65] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (May 2016), 83–89. https://doi.org/10.1109/MS.2015.83

[66] Juan Manuel Paniego, Silvana Gallo, Martin Pi Puig, Franco Chichizola, Laura De Giusti, and Javier Balladini. 2018. Analysis of RAPL Energy Prediction Accuracy in a Matrix Multiplication Application on Shared Memory. In *Computer Science – CACIC 2017*, Armando Eduardo De Giusti (Ed.). Springer International Publishing, Cham, 37–46.

[67] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL '02)*. Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[68] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. 2015. Deep face recognition.. In *BMVC*, Vol. 1. 6.

[69] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.

[70] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. 2011. Fine-Grained Power Modeling for Smartphones Using System Call Tracing *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 153–168. https://doi.org/10.1145/1966445.1966460

[71] Rui Pereira, Marco Couto, Joao Saraiva, Jacome Cunha, and Joao Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software (GREENS '16)*. ACM, New York, NY, USA, 15–21. https://doi.org/10.1145/2896967.2896968

[72] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[73] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 39–40.

[74] Richard N M Rudd-Orthner and Lyudmila Mihaylova. 2019. Non-Random Weight Initialisation in Deep Learning Networks for Repeatable Determinism. In *2019 10th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. https://doi.org/10.1109/DESSERT.2019.8770007

[75] Tara N Sainath, Brian Kingsbury, George Saon, Hagen Soltau, Abdel-rahman Mohamed, George Dahl, and Bhuvana Ramabhadran. 2015. Deep convolutional neural networks for large-scale speech tasks. *Neural Networks* 64 (2015), 39–48.

[76] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Commun. ACM* 63, 12 (Nov. 2020), 54–63. https://doi.org/10.1145/3381831

[77] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Commun. ACM* 63, 12 (Nov. 2020), 54–63. https://doi.org/10.1145/3381831

[78] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3645–3650. https://doi.org/10.18653/v1/P19-1355

[79] Marek Suchanek, Milan Navratil, Don Domingo, and Laura Bailey. 2018. 4.4. CPU Frequency Governors. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-cpufreq

[80] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[81] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101 _eprint: https://doi.org/10.3102/10769986025002101.

[82] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 683–693.

[83] Andy Wagner, Tiyasa Mitra, Mrinal Iyer, Godfrey Da Costa, and Marc Tremblay. 2020. Position Masking for Language Models. arXiv:2006.05676 [cs.CL]

[84] Yue Wang, Ziyu Jiang, Xiaohan Chen, Pengfei Xu, Yang Zhao, Yingyan Lin, and Zhangyang Wang. 2019. E2-Train: Training State-of-the-art CNNs with Over 80% Energy Savings. *Advances in Neural Information Processing Systems* 32 (2019). https://proceedings.neurips.cc/paper/2019/hash/663772ea088360f95bac3dc7ffb841be-Abstract.html

[85] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *IJCAI*. 3034–3040.

[86] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[87] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016).

[88] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 440–450.

[89] 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)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotedge18/presentation/zhang