Neural Computing and Applications (2024) 36:10297–10338
https://doi.org/10.1007/s00521-024-09588-z

ORIGINAL ARTICLE

A methodological framework for optimizing the energy consumption of deep neural networks: a case study of a cyber threat detector

Amit Karamchandani[1] · Alberto Mozo[1] · Sandra Gómez-Canaval[1] · Antonio Pastor[2]

Received: 9 April 2023 / Accepted: 5 February 2024 / Published online: 15 March 2024

Abstract
The growing prevalence of deep neural networks (DNNs) across various fields raises concerns about their increasing energy consumption, especially in large data center applications. Identifying the best combination of optimization techniques to achieve maximum energy efficiency while maintaining system performance is challenging due to the vast number of techniques available, their complex interplay, and the rigorous evaluation required to assess their impact on the model. To address this gap, we propose an open-source methodological framework for the systematic study of the influence of various optimization techniques on diverse tasks and datasets. The goal is to automate experimentation, addressing common pitfalls and inefficiencies of trial and error, saving time, and allowing fair and reliable comparisons. The methodology includes model training, automatic application of optimizations, export of the model to a production-ready format, and pre- and post-optimization energy consumption and performance evaluation at inference time using various batch sizes. As a novelty, the framework provides pre-configured "optimization strategies" for combining state-of-the-art optimization techniques that can be systematically evaluated to determine the most effective strategy based on real-time energy consumption and performance feedback throughout the model life cycle. As an additional novelty, "optimization profiles" allow the selection of the optimal strategy for a specific application, considering user preferences regarding the trade-off between energy efficiency and performance. Validated through an empirical study on a DNN-based cyber threat detector, the framework demonstrates up to 82% reduction in energy consumption during inference with minimal accuracy loss.

Keywords Software-defined networking · SDN controller · Cybersecurity · Machine learning · Deep learning · Energy efficiency

1 Introduction

Training deep neural network (DNN) models can be computationally expensive, requiring large amounts of memory and specialized hardware, such as GPUs or TPUs, which must run for several hours for typical applications, consuming a significant amount of power each time a new model needs to be trained. Additionally, when deployed on a large scale, DNN model inference can also consume a large amount of energy over the model lifetime, especially in real-time applications where buffering the data is not possible or is limited by the constraints of the environment. Furthermore, new instances of the model usually have to be created or destroyed on demand frequently, depending on the system load, which also adds to the overall high power consumption.

In addition to the above, the deployment of such networks also requires substantial use of the central processing unit (CPU), random access memory (RAM), and storage. The resources required by a DNN model have a direct impact on the number of models that can be simultaneously trained and deployed on a given server, as well as on the hardware specifications necessary to support these activities. Therefore, reducing resource utilization during runtime is imperative not only to mitigate the cost and carbon

✉ Alberto Mozo
a.mozo@upm.es

[1] E.T.S. de Ingeniería de Sistemas informáticos, Universidad Politécnica de Madrid, Calle Alan Turing s/n, 28031 Madrid, Spain

[2] Telefónica I+D, 28050 Madrid, Spain

footprint of DNN-based systems but also to improve their scalability and deployment flexibility. Although it is possible to increase the storage and resources of a system in response to changing requirements, this approach can result in higher power consumption, which leads to a critical trade-off between scalability, power, and performance. Moreover, the edge inference requirements are even more stringent, requiring devices capable of performing inference with precise latency, storage, computation, and energy consumption specifications.

To date, research in the area of energy-efficient DNN-based systems has focused mainly on energy-efficient hardware, such as custom DNN accelerators [1, 2], which aim to minimize the energy consumption of specific DNN architectures by reducing the number of operations performed and the amount of data transferred. However, these approaches are often not effective in reducing the energy consumption of DNN-based systems because they are limited to a handful of specific architectures, primarily convolutional neural networks, which are often not the architecture of choice for specific applications outside the visual domain. In addition, these approaches require that the hardware be equipped with the appropriate software, which can further limit the scalability and flexibility of a system. Furthermore, these techniques have not been widely adopted, partly because of the high cost of developing and maintaining custom hardware.

In contrast, other techniques that are completely agnostic to the architecture of the DNN model that is the target of the optimization process, such as model pruning [3] or weight quantization [4], have seen widespread adoption and are now used in many commercial products due to their ease of application and reasonable portability between hardware platforms. However, these techniques can potentially impair model performance, thus creating a trade-off between energy efficiency and model performance, which must be considered when deciding which optimization technique is best for a given application. Furthermore, changes in performance depend on the actual architecture of the DNN model and are, in general, unpredictable and difficult to control.

In general, optimizing the energy efficiency of a DNN-based system requires careful analysis of the trade-offs associated with energy efficiency and performance to determine the most appropriate optimization techniques for the application at hand. However, little research has been devoted to analyzing and understanding the trade-offs associated with energy efficiency and resource utilization in DNN-based systems. To our knowledge, this study is the first work to provide a methodological framework and an extensible experimentation platform for (i) evaluating the trade-offs associated with energy efficiency, resource utilization, and performance in DNN-based systems, and (ii) automatically selecting the most appropriate optimization techniques based on the application requirements (e.g., favoring performance over energy consumption or vice versa or balancing both).

This work proposes as a novelty an integrated methodological framework for analyzing and calculating the energy consumption and resource utilization of DNN-based systems in a production environment, which is implemented in an open-source extensible experimentation platform. The framework presents a new concept called "optimization strategies", which allows for the combination of several state-of-the-art architecture-agnostic techniques within the experimentation platform. Additionally, the framework introduces the concept of an "optimization profile" that effectively addresses the trade-off between energy savings and model performance. Profiles allow for quantitatively reflecting the expected balance between energy savings and model performance, allowing for the automatic selection of the most appropriate optimization strategy based on the specific user preferences and requirements of a given application.

In our framework, we consider two types of resources: energy, which is the main resource of interest, and computational resources, such as CPU, memory, and storage usage. The framework consists of the following four main steps: (1) a benchmark DNN model is trained using standard training techniques, without any compression or speedup; (2) a set of strategies designed by us and based on the combination of state-of-the-art model compression techniques, such as pruning, weight quantization, and knowledge distillation, are applied to the benchmark DNN model; (3) the resulting models are deployed in a production-ready format using an open-source DNN inference engine that is optimized for inference speed and resource utilization, such as TensorFlow Lite; (4) the energy consumption and resource utilization of the resulting models are measured in the production environment using performance profiling tools; and (5) the results obtained with the different optimization strategies are compared and the most suitable strategy is automatically selected according to the chosen optimization profile, producing the final optimized model ready to be deployed.

We demonstrate the effectiveness of our proposed framework by applying it to a complex real-world use case, namely cryptomining detection, which requires the use of highly efficient and high-performance DNN models for real-time classification of network traffic with high accuracy due to the critical nature of the application [5]. In particular, we validate our proposed framework in a realistic network scenario by successfully applying it to optimize a DNN classifier deployed in a cybersecurity component of TeraFlowSDN, an open-source cloud-native software-defined network (SDN) controller [6], in which

the cybersecurity component is responsible for monitoring and detecting cryptomining activity in the network [7]. The results obtained by the optimization framework show that the optimized DNN-based detector achieves an 82.304% reduction in total energy consumption while maintaining a 0.008% loss in the target performance metric (balanced accuracy). We provide an in-depth analysis of the energy consumption, resource utilization, and performance of the resulting models and how they are affected by the choice of optimization techniques.

Our research offers three key contributions. First, we present an integrated methodology and a flexible experimentation platform that enables accurate analysis of energy efficiency, performance, and resource utilization in DNN-based systems operating in production environments with strict computational and scalability requirements. The experimentation platform is based on a versatile design and can be easily adjusted to incorporate new optimization techniques and evaluation metrics, making it suitable for a wide range of applications. Moreover, the platform incorporates several adaptive algorithms that can automatically determine and implement the optimal combination of optimization techniques required to meet specific application requirements through a rigorous examination of the trade-off between energy efficiency and model performance exhibited by the target model. Second, in an effort to enhance the academic community's understanding of the various approaches available for energy optimization of DNNs, our work provides a comprehensive and critical review of the current state of the art of model optimization techniques. Finally, we demonstrate the practical application of our methodology by evaluating the trade-offs between energy efficiency, performance, and scalability in a cryptomining detector, a complex use case deployed in a real-world network production environment.

1.1 Contributions

In this section, the main contributions of our work are summarized below.

- We propose a novel methodology to reliably analyze the energy efficiency and resource utilization of DNN-based systems when deployed in realistic production environments.
  - The methodology is implemented in a publicly available open-source framework that contains a DNN inference engine optimized for inference speed and resource utilization.
  - The methodology proposes as novelty a set of strategies based on a combination of state-of-the-art model optimization techniques.
  - As an additional innovation, several optimization profiles adapted to different application scenarios (e.g., favoring performance over energy consumption or vice versa or balancing both) are also proposed in the framework.
  - In addition, a wide variety of model optimization techniques and resource utilization and performance metrics are supported out-of-the-box, which makes our framework suitable for a wide range of applications, and it is also easily extensible to adapt to specific needs.

  The proposed framework can be a valuable tool for researchers and practitioners, providing a complete scientific basis for the design, optimization, and evaluation of novel model optimization techniques.
- The effectiveness of our proposed framework is demonstrated by applying it to a real-world use case subject to real-time constraints and high scalability requirements, namely cryptocurrency mining detection. In particular, we successfully applied the proposed framework to optimize a DNN classifier deployed in a SDN controller responsible for monitoring and detecting cryptocurrency mining activity in the network. We provide a systematic in-depth analysis of the energy consumption, resource utilization, and performance of the resulting models, along with a critical analysis of the trade-offs found. To the best of our knowledge, this is the first work that provides a comprehensive analysis of the trade-offs between energy efficiency, resource utilization, and model performance obtained with a variety of state-of-the-art machine learning (ML) and deep learning (DL) model optimization techniques for a realistic use case with ML/DL models deployed in a real-time environment.
- The source code for our proposed framework is released, in order to promote wider adoption and further research in this field. The open-source nature of our framework provides transparency and reproducibility, facilitating replication and promoting the extension of research in this field. In addition, the release of our framework can foster collaboration and sharing of ideas and techniques, leading to a more diverse and comprehensive approach to energy optimization and evaluation of DNNs, especially in production environments with high scalability requirements and real-time constraints. By providing access to our methodology, we hope to contribute to the advancement of the field, serving as a starting point for future research in the area, and allowing researchers to build upon our work and develop new techniques and methodologies to advance in the energy optimization and evaluation of DNNs. Access to the repository containing the framework's source code, installation

instructions, documentation, example code, and data is available at github.com/amitkbatra/EnergyNet.

1.2 Paper structure

The remainder of this article is organized as follows. In Sect. 2, we discuss related work in the area of energy efficiency and resource utilization analysis of DNN-based systems. In Sect. 3, we describe the main model optimization techniques that have been proposed to improve the energy efficiency of DNN-based systems. In Sect. 4, we describe our proposed methodological framework for analyzing the energy efficiency and resource utilization of DNN-based systems when deployed in production environments. In Sect. 5, we present a complex real-world use case, namely cryptomining detection, and the DNN-based system that we implement in an SDN controller to detect cryptomining activity in real time by means of passive monitoring of network traffic. More specifically, we describe the benchmark model we used as the basis for our energy efficiency analysis. In particular, we provide details on the dataset created to train the model, we describe the model architecture and the training procedure, and we evaluate the model's performance. In Sect. 6, we demonstrate the effectiveness of our proposed framework by applying it to the cryptomining detection use case. In particular, we analyze the energy consumption, resource utilization, and performance of the resulting models and how they are affected by the choice of optimization techniques. Section 7 provides an integrated discussion on the main findings of the experimental evaluation, as well as on the scalability aspects of the framework and its current limitations with possible future research directions. In Sect. 8, we summarize our findings and discuss future work.

2 Related work

This section provides a comprehensive review of the current research on evaluating the energy efficiency of DNNs, identifying significant gaps that this article aims to address.

Authors in [8] investigate the financial and environmental expenses associated with training large data-driven DNN models for natural language processing (NLP). The authors quantify the approximate costs of training various recently successful DNN models for NLP and propose actionable recommendations for reducing costs and enhancing equity in NLP research and practice.

In [9], an extensive analysis of the metrics crucial for the practical implementation of DNNs in computer vision is presented. Specifically, the article evaluates various DNNs in the ImageNet classification challenge on metrics such as accuracy, memory footprint, parameters, operation count, inference time, and energy consumption. The study also provides significant findings related to the interdependence of these metrics, such as the hyperbolic relationship between accuracy and inference time and the influence of energy constraints on accuracy and model complexity. Furthermore, the article discusses how more efficient architectures can be utilized to produce more compact representations than current models.

The study presented in [10] investigates the power and energy efficiency of convolutional neural networks (CNNs) when running on CPUs and GPUs. The article provides an in-depth analysis of several well-known DL frameworks, examining the impact of hardware settings on performance and energy efficiency. The authors measure the power consumption of different networks and layers and compare the performance of native implementations of neural network operations on GPUs with those present in the cuDNN library. Furthermore, the study evaluates the performance of the Caffe DL framework and its derivatives on the CPU, exploring the performance of various libraries such as Atlas, OpenBLAS, MKL, and Caffe-OpenMP.

Authors in [11] survey different approaches to model power and energy consumption of ML algorithms. First, the authors provide general information regarding the fundamentals of power and energy consumption. Additional clarifications on how software programs consume power are also described. In addition, they also provide an overview of the various methods available for estimating the proposed energy consumption, classified by type (analytical or empirical measurement), technique (hardware-based monitoring or simulation), and level (instruction or application). In addition, the authors describe how these power and energy consumption estimation methods can be adapted to specific ML scenarios based on three key characteristics: target, dataset size, and training type (offline or online learning). The authors also present a case study to illustrate the mapping suggestions. It provides an example of how the suggested methods can be applied to measure energy consumption in a real-world scenario.

A comprehensive literature review of energy estimation approaches, grouping the surveyed works into a high-level taxonomy can be found in [12]. The authors describe the advantages and disadvantages of each category to guide the selection of the most appropriate energy consumption estimation method for specific application scenarios. Additionally, the authors offer a detailed explanation of all reviewed articles and provide an overview of current research in ML concerning energy and power estimation, with a focus on the DL field. The article also describes currently available software tools for building power consumption models and their features, as well as two interesting use cases of energy consumption evaluation of ML models, one focused on data stream mining and the other

on inference with CNNs. In the case of data streaming, the authors conclude that there is a trade-off between energy efficiency and performance of the evaluated techniques.

The authors suggest taking this trade-off into account when selecting a specific ML model for a given application scenario, optimizing accuracy or energy efficiency based on available battery life. Finally, the authors evaluated the energy consumption of a CNN in a second use case, obtaining detailed energy estimates of different layers and applying a regression-based approach to performance counter information to predict the energy consumption of the convolutional layers, which was subsequently validated with real energy measurements.

To the authors' knowledge, no other study has provided a complete analysis of the computational requirements of DNN models during training, inference, and loading stages using a comprehensive set of state-of-the-art optimization techniques. Additionally, our study introduces a novel systematic approach to measuring the balance between energy efficiency and model performance, enabling the automatic selection of the most effective optimization strategy based on the application's specific needs.

3 Energy efficiency optimization techniques for deep neural networks

DNN training is often performed on expensive hardware accelerators that lead to high energy consumption and carbon emissions [13]. This issue has been a growing concern in the AI community in recent years. It has motivated the development of more efficient algorithms to reduce the environmental impact of ML/DL models. This set of techniques, commonly referred to as Green AI, aims to fulfill this promise by reducing the energy required in the training and inference of ML/DL models.

A common approach to achieving Green AI is to use more efficient hardware. Some studies have focused on developing application-specific accelerators based on FPGA or ASIC technology to efficiently train arbitrarily sized DNNs to optimize a given energy estimation model [1, 2]. Although custom hardware optimized for DNN can offer significant improvements in efficiency over general-purpose hardware, it is often expensive and challenging to develop. Architectural designs must be handcrafted to optimize specific DNN architectures, undoubtedly a laborious and time-consuming task. Although some frameworks have been proposed to alleviate this issue [14, 15], they are fundamentally confined to convolutional neural networks (CNNs); therefore, their general applicability to other architectures remains unsolved. The hardware capabilities that these devices provide are another challenge. The relatively low frequency at which FPGA and ASIC boards operate and the small on-chip memory they provide drastically limit their adoption for accelerating DNNs in contexts beyond edge computing scenarios.

In addition to hardware optimization, another practical approach to realizing Green AI is to use more efficient algorithms in their use of resources. This can be done by reducing the number of computations required for training, using more efficient data structures, or finding ways to reuse computations. However, prior to the optimization of the ML model, it is imperative to first establish accurate methods to estimate the energy cost associated with training or inference tasks. Several studies have focused on optimizing ML-specific models by adopting general estimation models that characterize energy consumption (either at the software or hardware level) in the ML domain. However, most of these studies specifically target battery-constrained devices [16] or rely on simplistic approaches that extrapolate hardware measurements to evaluate the energy cost associated with each operation [17, 18] or relay on indirect measurements (i.e., proxies) to determine memory consumption such as number of floating point operations (FLOPS) [19] or memory access counts [17]. In addition, it must be noted that there is a lack of studies that consider the design of GPU-aware energy estimation models. As an exception, [18] proposes a method for a priori evaluation of the power consumption required for CNN model training on GPU that combines application-level characteristics, such as per layer number of kernels and kernel size, with the power consumption of the GPU device that is associated with the application. This study has shown that it is feasible to obtain a high correlation between predicted measurement and actual consumption that is sufficiently accurate and that can also be used to predict the power consumption of the training task in advance. However, as in most current studies, this method is specifically designed to estimate the cost of a specific DNN architecture, and therefore its scope of application is minimal. To keep pace with the rapid pace of scientific advances in the design of DL model architectures and emerging generational hardware improvements, future research should integrate GPU consumption into estimation models to provide accurate but also architecture-agnostic models, either using hardware component measurements or via hardware simulators.

One way to reduce the energy consumption of DNN is to use neural architecture search (NAS). NAS is a method for automatically identifying the best DNN architecture for a given task. There are many different ways to perform NAS, such as NASNet [20], DARTS [21], AmoebaNet [22], AutoKeras [23], ENAS [24] or ProxylessNAS [25], and the best method for a given application will depend on the specific constraints and objectives. More importantly, to find DNN architectures that are accurate and energy-

efficient using NAS, energy estimation must be incorporated as another optimization objective during the training procedure. Unfortunately, very few studies have emerged to accomplish this task. The proposed approaches range from low-rank factorization [26, 27] to Bayesian optimization [28] or genetic algorithms [29].

Another approach is to split and distribute the computation of DNN training across multiple devices. Distributed training techniques, such as federated learning, can offer significant efficiency improvements, as they allow for harnessing the computational power of multiple devices to perform the computation. However, it can also be more difficult to deploy, as it requires careful coordination of the different devices. Moreover, modern distributed training algorithms are designed for HPC clusters with high-bandwidth communications between powerful GPUs. To make distributed training efficient for larger networks, some techniques have been proposed to reduce the communication bandwidth. For example, [30] proposed layer-wise adaptive rate scaling (LARS), a method for training DNN efficiently with large batches, reducing the number of information exchanges through the communication link. Other techniques such as deep gradient compression [31], PowerSGD [32], and 1-bit Adam [33], to name a few, have been proposed to compress gradients and thus reduce the required communication bandwidth and the training latency. On the other hand, parameters between model layers can be reused, resulting in a model with fewer parameters. In this way, fewer gradients need to be exchanged through the network, thus resulting in more efficient network utilization. [34, 35] proposed parameter-sharing techniques for NLP and computer vision.

Model compression (MC) is a related approach to more efficient DNN training. This technique aims to reduce the size of the DNN model, which can lead to improved efficiency. MC can be done by pruning unused weights, using lower precision weights, or using more efficient data structures. Pruning is a technique to reduce the size of a DNN by removing unnecessary weights [3]. Pruning can be performed manually or automatically. Manual pruning involves removing weights from a neural network by applying a manually designed heuristic approach. This can be done by removing individual weights or removing entire network layers. Automatic pruning involves using algorithms to automatically remove unnecessary weights from a DNN. This can be done by training a smaller DNN and then using that network to prune the larger network.

Moreover, quantization is a technique that allows DNN models to be trained or used for inference with lower precision, leading to improved efficiency [36]. [37] successfully applied stochastic quantization to convert gradients to lower bit-width representations during the backward pass. This enables the use of bit convolution kernels during the backward and forward passes, which further accelerates training times and speeds up inference times. Quantization has also been applied in the form of a differentiable nonlinear activation function to reduce training times [38]. In this way, the quantization is learned in a lossless and end-to-end fashion during the training. This method is suitable for arbitrary bit-width quantization and can be applied to both weights and activations of the DNN.

Another technique is low-rank factorization (LF), which is a technique for representing a matrix with a smaller number of parameters. This can be done by decomposing the matrix into a product of two lower-rank matrices. Taking advantage of this concept, LF has been applied to reduce weight matrices that represent the parameters of the ML / DL model [39]. LF has also been applied to the compression of training datasets [26, 27]. In DNN, the number of neurons in the input layer depends on the size of the feature space. By reducing the dimensionality of the feature space, the size of the DNN is also reduced. To achieve this reduction in dimensionality, input data in the form of matrices are factorized into low-rank matrices to adjust to the hardware characteristics in an automated fashion. In this way, both the in-memory size of the DNN and the inference time are reduced, while the accuracy is preserved.

On the other hand, knowledge distillation (KD) is a technique for transferring the knowledge learned by a large DNN into a smaller one. This can be done by training the smaller DNN to mimic the predictions of the larger one [40]. KD, which was first introduced by [41], has been successfully applied to compress large transformer models such as BERT (distilBERT) [42].

In addition, transfer learning (TL) is a technique that can reduce the amount of data and computing power required to train a model. TL involves using a model that has already been trained on a similar task to train a new model. This can be done by fine-tuning the weights of the pre-trained model or by using the pre-trained model as a feature extractor. Finally, the hyperparameter tuning process for DNNs is computationally expensive and is often the main bottleneck in model training times, especially for those with billions of parameters. Therefore, poor hyperparameter selection can lead to poor performance and potential waste of resources (increased training times and energy consumption). Related to the above, recently, [43] proposed a new method to reduce the cost of hyperparameter tuning for DNNs. This method is based on the discovery of maximum updating parameterization, a variation of the standard hyperparameter fitting process that demonstrates that when hyperparameters are scaled in a specific way, they can be zero-shot transferred from a (narrower) proxy model to a (large width) target model with approximately optimal performance, thus eliminating the need to fit the

target model at all. Note, however, that regularization parameters cannot be adjusted with this technique, so it is only considered beneficial when regularization is not the bottleneck of training. Also, since regularization is an essential part of fine-tuning tasks for pre-trained models in TL, this technique is inappropriate for these use cases. Furthermore, the authors state that some hyperparameters cannot be transferred when the proxy model is trained with different data or for a different task, which further reduces its utility for fine-tuning processes.

In summary, the main approaches to improving the energy efficiency of DNN include:

1. Use of specialized hardware specifically designed to accelerate the computation of specific DNN architectures and provide significant performance improvements over the use of commodity CPUs or GPUs.
2. Reducing the number of neurons and connections in the network using model pruning or knowledge distillation, thereby reducing the amount of computational resources required to run the network without significantly degrading its performance.
3. Use of low-precision data types to represent weights and network activations, making calculations more efficient.
4. Use of energy-efficient distributed training techniques to leverage the computational power of multiple machines to reduce the time required to train the network.
5. Using transfer learning to reuse the knowledge gained by existing models and reduce the training time and computing resources required to obtain satisfactory performance of new models.

4 Methodology and design of the proposed machine learning energy efficiency optimization framework

In this section, we describe the experimental framework of our proposed methodology for analyzing the energy efficiency and resource utilization of DNN-based systems deployed in production environments. First, we provide an overview and a visual representation that illustrates the main steps of our proposed methodology. Then, we explain the energy measurement process that our framework uses to collect energy consumption data and discuss the main statistics it collects to analyze the resource utilization of the resulting models. Next, we describe how the most appropriate optimization strategy for the problem at hand is selected in our framework. After that, we describe the energy optimization techniques supported by our framework. Finally, we describe the software stack that we used to implement the proposed methodology and the requirements that the hardware platform must meet to properly run the framework.

4.1 Overview of the proposed methodology

In Fig. 1, we represent the complete workflow of the proposed methodology. As can be seen, the methodology is divided into two phases: an optimization phase and an analysis phase. The first step is to train the model to be optimized and that will serve as the baseline for the evaluation. The second step is to apply a set of energy optimizations such as pruning or weight quantization techniques to our target DNN. In the third step, the model is converted to a production-ready format to obtain a faithful evaluation of the energy consumption and performance of the model once it is deployed. In the fourth step, the energy consumption and resource utilization profiling of the DNN during the inference (using different batch sizes) and loading stages is performed. In the fifth step of the methodology, the model is evaluated using different metrics to compare its performance before and after optimization. Finally, in the sixth step, the energy-accuracy trade-off of the non-optimized model and the one that results from the application of different energy optimizations is analyzed to provide a detailed report of the energy efficiency gain and the impacts on the prediction accuracy of the target DNN that is obtained with each technique. In the seventh and final step, the automatic selection of the best optimization strategy is carried out according to a user-defined optimization profile that represents the user's preferences in terms of the relationship between energy efficiency and accuracy shown by the final optimized model.

4.2 Measuring energy, performance and resource consumption

Measuring the energy consumption of a computer program is a complex task due to the multiple factors involved in this process, such as the computer architecture, the execution environment, and the different power modes of the computer or battery usage, among others. Therefore, there is no precise procedure to measure the energy consumed by a program in a deterministic way, since different computers can behave very differently depending on their components, the program to be executed, the operating system, and even the environmental conditions. The same program may produce different results in energy consumption on different computers. For this reason, power consumption can only be estimated by measuring the resource consumption and power usage of a group of representative executions and applying a statistical average to aggregate

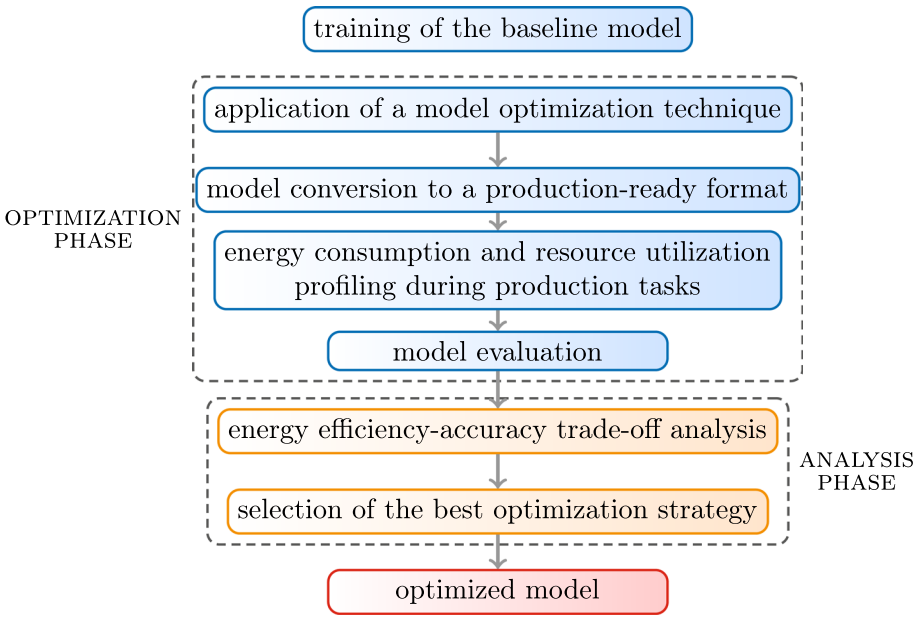

Fig. 1 Summary of the main steps of the workflow proposed in our energy efficiency methodological framework for DNNs

them to produce an estimate of the real power consumption.

There are several ways to estimate the energy consumption of a program [11]. Traditional approaches rely on empirical evaluation of power consumption using dedicated power meters placed at different locations on the computer (power supply, GPU, VRM, etc.). However, this approach lacks precision, as it cannot be used to estimate the power consumption related to the execution of a particular program. More fine-grained approaches consist of estimating power consumption using simulators that model the behavior of the main components of a computer to obtain an estimate of the power consumption of each component. The advantage of this method is that it offers the possibility to easily instrument a program with monitoring code that can collect measurements of the usage of each component at different execution levels [12]. However, simulators are often not able to capture a wide variety of factors that influence a program's energy consumption due to the high complexity and computational power required to accurately model the behavior of the various components of a computer and their interactions. In fact, the main drawback of this simulation-based approach is that capturing many aspects of the processor's internal behavior that greatly influence a computer's power consumption, such as cache, memory, bus, registers, pipelines, and memory mapping, can be very complex to simulate properly and accurately. However, attempting to simulate them accurately places a large computational burden that makes it difficult to simulate complex programs with a large instruction set or program loops accurately. Therefore, this approach does not produce satisfactory results for highly parallel and complex computer programs, such as programs with many execution paths, branches, jumps, etc., or using specialized instructions, as is precisely the case for ML/DL models. Moreover, the induced overhead of simulator-based approaches leads to the impossibility of measuring energy use in real-time throughout the execution of a program at a high level of resolution [11], which is essential for a correct and reliable evaluation of the energy consumption of a program.

Performance monitoring counters (PMCs) are hardware-based monitors that are integrated into the processor or a specific processor chip to collect data on specific microarchitecture-related events that occur during computer execution [12]. PMCs include event counters that count the number of occurrences of certain hardware events (such as cache access, instructions per cycle, etc.). PMCs are generally considered reliable because they can monitor every access to the computer's core and memory, and thus provide very accurate information on power consumption in real-time and with low-performance overhead. Importantly, the real-time measurements provided by PMCs allow for dynamic energy optimizations based on current energy consumption [11], which is highly useful in certain scenarios (real-time or near real-time inference, online learning, etc.). Many researchers have taken advantage of PMCs to develop energy models capable of estimating computer power consumption, even at the application level. In this regard, Intel has developed a state-of-the-art energy model called RAPL, which estimates power consumption based on the PMC values that can be collected from Intel family processors. The framework proposed in this study uses the RAPL interface through the powerstat command line profiling tool to collect the CPU

power consumption of the ML models during the training, model optimization, inference, and model loading phases.

In addition to energy measurement, information on resource consumption is collected during the three phases of the ML process (training, model optimization, and model loading). In the following, we will refer to each of these three phases as *a test*. To collect these measurements the psutil (python system and process utilities) library is used, which is a cross-platform library for obtaining information about running processes and system resource usage (CPU, memory, disks, network, sensors) in Python. In particular, psutil is used to collect CPU and RAM usage, as well as CPU frequency and disk IO statistics. The reason we chose to use psutil is that it is free, open-source, cross-platform, and capable of collecting the information needed for our case. We decided to collect CPU and RAM metrics because they have proven useful for other similar studies [44] and will be especially useful in this study because ML algorithms have been shown to have a large RAM footprint and also require many processing cycles during their loading, training, and inference.

Resource consumption measurements will be used to provide information on the computational requirements of ML models with different optimizations applied. In this way, the computational needs of performing training and inference tasks with different optimizations applied can be determined, which will allow adjusting the hardware configuration where the models will be deployed to maximize resource utilization and minimize energy expenditure by avoiding over-provisioning of hardware components and idle power usage. In addition, these measurements provide interesting information on scalability, as they can help determine the number of instances that can be called on demand to maximize resource utilization.

For each combination of techniques to be applied, the baseline model, which, in our case, is the one evaluated in Sect. 5.2.2, is trained and then the optimization techniques are applied sequentially depending on the order specified in the optimization strategy defined by the particular combination to be applied.

Once the model has been trained and optimized, its inference performance is evaluated. To evaluate the inference performance of the model obtained with each combination of techniques, the model is converted from TensorFlow/Keras to TensorFlow Lite format. TensorFlow Lite is one of the most common platforms for deploying ML models in production. Being fully open source, cross-platform, and offering an industrial-grade solution, it is an optimal choice for most ML applications. Considering that the inference phase is the most energy-consuming in ML applications, converting the model to TensorFlow Lite format to speed up and optimize the inference process has proven to be a great gain in terms of both energy and performance. For this reason, TensorFlow Lite and other similar solutions such as open neural network exchange (ONNX) have been widely adopted in the industry to deploy ML models in production. Converting the TensorFlow model to a production-ready format such as that offered by TensorFlow Lite is therefore an essential step in providing a realistic assessment of the energy consumption of ML systems in the inference phase. Note that other alternatives, such as ONNX, were discarded because of the immaturity and high complexity of this tool when working with advanced optimization techniques, such as the one applied in this work.

One of the disadvantages of using TensorFlow Lite is that it does not yet provide support for GPU inference. Therefore, to evaluate the performance of the model on GPU, it is necessary to export the model to other frameworks that provide this support such as TensorRT. However, since this time we will focus our evaluation on CPU-based deployments, being the most used in production systems due to their lower cost and high scalability and flexibility compared to GPU solutions, the evaluation of the model will be performed on CPU platforms using TensorFlow Lite. In the future, we plan to extend our framework to add support for GPU inference using TensorRT or other similar frameworks.

In the inference stage, various batch sizes are tested to evaluate the variation in power consumption and resource utilization. By default, the small (32), medium (256), and large (1024) batch sizes are tested, although they can be configured by the user depending on the application requirements.

In addition, the predictive performance of the optimized ML model obtained with each combination of techniques is also measured. Our framework supports and generates a complete list of performance metrics generally used to evaluate the performance of classification and regression models. More specifically, in the case of optimizing a binary classifier, the following metrics are used: accuracy, precision, recall, F1-score, and AUC, as well as balanced accuracy and Matthews correlation coefficient to account for the class imbalance problem. On the other hand, for regression tasks, the following metrics are used: mean absolute error, mean squared error, mean absolute percent error, and symmetric mean absolute percent error.

Depending on the particular task that needs to be solved, our framework will select the most appropriate metrics to perform the evaluation. The user must determine the type of task to be solved and select a metric as the objective to be optimized during the selection of the most appropriate combination of techniques to be applied, which is discussed in the next section.

In addition, our proposed framework can be extended to incorporate arbitrary performance metrics which will be

systematically computed and collected during the execution of all optimization strategies, ensuring a comprehensive performance evaluation.

For each of the optimization strategies that can be applied, a configurable number of repetitions for each test (model training, inference, and load) can be defined (by default is set to 5) together with an intermediate waiting time to restore the initial conditions of the machine where the framework is being executed. By setting a high number of repetitions, more reliable measurements can be obtained. During each repetition, the metrics listed hereafter are collected at a configurable time interval (default is set to 1 s).

- Unique Set Size (USS) of the process.
- Percentage of CPU used by the process.
- System-wide CPU frequency.
- System-wide CPU power draw.

Other metrics such as the amount of physical and virtual memory used by the process, as well as the number of I/O operations performed by the process during the test, can also be measured. However, these metrics have been discarded due to the low variability shown in the measurements obtained during the execution of the framework, showing very similar values for all the tests performed. For this reason, they are considered irrelevant to evaluate the difference in resource utilization of the optimized and non-optimized models and, therefore, have been discarded since their inclusion would have meant an increase in the computational load of the framework with little or no added value.

The reason we decided to collect system-wide CPU power consumption instead of process-specific CPU power consumption is due to the unavailability of tools that allow us to collect this type of information reliably and accurately. In any case, system-wide CPU usage remains a perfectly valid and reliable measure, as long as the user ensures that no other CPU-intensive background computations are performed during the execution of the framework, which may bias the results. For this reason, the measurements obtained can be perfectly treated as a rough estimate of the amount of CPU power consumed in each test.

Once all repetitions have been performed, the metrics obtained at each time step among all repetitions are aggregated using the mean, standard deviation, and maximum value. In addition, a second aggregation is also performed, but on this occasion on the time axis to show the mean value, standard deviation, and maximum of each statistic measured throughout the test as a summary of the results. At this point, the total energy consumed in the test is obtained by multiplying the average energy consumption by the average duration of the test. Furthermore, the percentage of reduction in total average energy consumption is also calculated by computing the difference between the average total energy consumption of each test with that obtained for the same test performed with the baseline model and dividing the result by the average total energy consumption of the test performed with the baseline model. In this way, a percentage is obtained that can be used as a metric of the improvement in energy efficiency achieved by the proposed optimizations. The obtained value can be positive or negative depending on the change in the energy consumption of the optimizations with respect to the baseline. A positive value shows that the optimizations perform better than the baseline, while a negative value shows that the optimizations consume more energy than the baseline.

On the other hand, for each test, the size of the compressed model disk is also reported using the Deflate algorithm. Compression is especially important for pruned models because in the serialized weight matrices the pruned connections are represented as zero and therefore have the same size as the weight matrices of the non-pruned model. Therefore, a compression algorithm has to be applied to remove this redundancy and obtain a much smaller model. For this reason, this step is crucial to obtain a representative file size of the ML that is deployed in production.

4.3 Selection of the best optimization strategy: optimization profiles

To select the most appropriate combination of optimization techniques to apply for a given target model, an exhaustive search of all optimization strategies is initially performed. More specifically, the framework builds a set of all possible combinations of optimization techniques that can be applied to the target DNN model. Once the set of all combinations has been constructed, the framework will evaluate each combination in the set by measuring the energy consumption, performance, and resource utilization as was previously defined in Sect. 4.2.

After a rigorous evaluation process of the results obtained with all optimization strategies, the framework will automatically identify and apply the optimization strategy that best aligns with the user's prioritized objectives for the specific task at hand. To this end, we introduce the concept of optimization profile to guide the optimizer in selecting the optimal strategy for a particular application according to the level of importance of energy consumption reduction versus potential performance loss when optimizing the target model. In this way, different optimization profiles can be defined to weigh a more aggressive fit in terms of energy reduction versus model performance and vice versa. Therefore, before starting the

model optimization process, the user must select the optimization profile to be used. In particular, we propose three different optimization profiles, although additional profiles can also be explored in the future.

1. *Energy efficiency profile* This profile is designed to minimize as much as possible the energy consumption of the optimized model while providing acceptable performance for the user. For this purpose, all applied optimization strategies are ranked in descending order according to the reduction in energy consumption they provide compared to the non-optimized model, and then the optimization strategy or strategies that provide the largest reduction are selected. If several optimization strategies provide the same reduction, the optimization strategy that provides the highest performance among them is chosen. The performance of the optimized model, as measured by a user-defined performance metric, is compared to the user-defined threshold. If the performance is below the threshold, the next optimization strategy that provides the greatest reduction in energy consumption is selected, and so on until the threshold is reached or all optimization strategies have been evaluated. If no optimization strategy provides acceptable performance, the non-optimized model is returned to the user and the framework will report the result obtained for each of the optimization strategies tested so that the user can decide which optimization strategy provides the best balance between performance and energy efficiency for their particular case. Alternatively, the user can select the balanced profile, described below.
2. *Performance profile* This profile is designed to maximize the performance of the optimized model while providing an acceptable energy efficiency gain with respect to the non-optimized model. To do this, all applied optimization strategies are ranked in descending order of the performance they provide based on a user-defined target performance metric, and then the optimization strategy or strategies that provide the highest performance are selected. If several optimization strategies provide the same performance, the optimization strategy that provides the highest reduction in energy consumption among them is chosen. The reduction in energy consumption of the optimized model relative to the non-optimized model is then compared to the user-defined threshold. If this relative reduction in energy consumption is less than the specified threshold, the next optimization strategy that provides the highest performance is selected, and so on until the threshold is reached or all optimization strategies have been evaluated. If no optimization strategy provides an acceptable reduction in energy consumption with respect to the non-optimized model, the non-optimized model is returned to the user, and the framework will report the result obtained for each of the optimization strategies tested so that the user can decide which optimization strategy provides the best trade-off between performance and energy efficiency for their particular case. Alternatively, if the performance profile cannot be satisfied, the user can select the balanced profile, described below.
3. *Balanced profile* This profile is designed to balance performance and energy efficiency by applying the optimization strategy that provides the best trade-off between both metrics. For this purpose, two parameters are provided: the weight of the performance metric and the weight of the energy efficiency metric. The weight of each metric indicates the importance that the user gives to that metric when selecting the optimization strategy to be applied. For example, if the user wants performance to be twice as important as energy efficiency, the weight of performance would be set to 1 and the weight of energy efficiency to 0.5. On the other hand, if the user wants performance to be the most important metric, the performance weight would be set to 1 and the energy efficiency weight to 0. Similarly, if the user wants energy efficiency to be the most important metric, the performance weight would be set to 0 and the energy efficiency weight to 1. The energy efficiency and performance gain metrics are normalized to have a range of 0 to 1. For both metrics to have the same optimization direction, the energy efficiency is inverted, so that the lowest possible value is assigned to one, and the highest possible value is assigned to zero. Then, both values are multiplied by the weight of each metric and summed. The optimization strategy with the highest score is used to obtain the final model.

A more concise explanation of each optimization profile is provided below:

1. *Energy Efficiency Profile* Select the optimization strategy, O, that minimizes energy consumption, E, while providing acceptable performance, P, according to the following procedure:

$$O_e = \arg\max\{E(o) \mid o \in \mathcal{O}, P(o) \geq P_{threshold}\}$$

   where $P_{threshold}$ is the user-defined performance threshold. If no optimization strategy meets the threshold, return the non-optimized model.
2. *Performance Profile* Select the optimization strategy, O, that maximizes performance, P, while providing acceptable energy efficiency gain, E, according to the following procedure:

$$O_p = \arg\max\{P(o) \mid o \in \mathcal{O}, E(o) \geq E_{threshold}\}$$

where $E_{threshold}$ is the user-defined energy efficiency threshold. If no optimization strategy meets the threshold, return the non-optimized model.

3. *Balanced Profile* Select the optimization strategy, O, that balances performance, P, and energy efficiency, E, according to the following procedure:

$$O_b = \arg\max\{w_P \cdot P(o) + w_E \cdot (1 - E(o)) \mid o \in \mathcal{O}\}$$

where w_P and w_E are the user-defined weights for performance and energy efficiency, respectively. The values for performance and energy efficiency are normalized to have a range of 0 to 1, and the energy efficiency is inverted so that both metrics have the same optimization direction. The final score is the sum of the performance and energy efficiency scores, weighted by the user-defined weights.

4.4 Framework requirements

The framework was implemented using Python (version 3.10.6) as the main programming language and using the following dependencies: TensorFlow (version 2.9.2), TensorFlow Model Optimization (version 0.7.3), psutil (version 5.9.4) and powerstat (version 0.02.27).

To run the proposed framework correctly, it is necessary to have a Linux operating system with a CPU that supports the Intel RAPL interface. If the CPU does not support this interface, the powerstat library will not work properly and the total energy consumed by the DNN during training and inference will not be obtained. As a result, only the resource utilization of the DNN during the training, inference, and load stages will be obtained.

4.5 Supported model optimization strategies

In this section, first, we explain the initial selection of a comprehensive set of several state-of-the-art optimization techniques that have been integrated into the framework and were previously presented in Sect. 3. Next, each of these techniques will be systematically explained, encompassing both their theoretical foundational principles and practical applications within the context of our proposed framework.

4.5.1 Overview of the selected optimization techniques

The initial repository provided "out-of-the-box" in the framework covers a wide spectrum of DNN optimization techniques, including quantization-based methods, pruning-based methods, transfer learning-based methods, metaheuristic approaches, and hybrid approaches based on combinations of different types of techniques. A brief summary of the three sets of techniques initially included in our framework is provided below.

- *Post-training optimization techniques* These techniques focus on alterations that are applied after the model has been trained. They include approaches such as full 8-bit integer (INT8) weight quantization, half-precision floating-point (FP16) weight quantization, and full integer weight quantization with 16-bit integer (INT16) activations and 8-bit integer (INT8) weights.
- *Training-aware optimization techniques* These strategies are applied by fine-tuning the trained model. They involve methods like pruning-aware model fine-tuning, quantization-aware model fine-tuning, and a combination of neural architecture search coupled with knowledge distillation.
- *Combined optimization techniques* Combining multiple strategies, these techniques involve hybrid approaches such as pruning-aware model fine-tuning combined with quantization-aware model fine-tuning, neural architecture search combined with knowledge distillation and pruning-aware model fine-tuning, among others.
- *Metaheuristic optimization* A metaheuristic method is specifically engineered to search for the optimal model taking into account both the measured energy efficiency and the exhibited performance. It operates as an adaptive mechanism, utilizing a combination of parameters and performance indicators to iteratively navigate the model space, optimizing both energy efficiency and performance metrics simultaneously. This metaheuristic method is applied in synergy with the knowledge distillation technique and also as a mechanism to automatically search for the overall optimal model among the various optimization strategies applied according to the optimization profile that was selected for the optimization process, as discussed in Sect. 4.3.

This initial selection of optimization techniques was oriented toward establishing a robust basis for comparison within the study and showcasing the framework's ability to automatically enhance energy efficiency. The intention was to demonstrate various widely recognized within the field for their efficacy in optimizing the inference process of DNN models (see Sect. 3), emphasizing the versatility of our proposed framework in accommodating diverse model optimization paradigms.

The framework was purposefully designed to be open source, emphasizing its extensibility by providing a well-documented structure for easily integrating new optimization techniques. This adaptability allows for the inclusion of both current and future advancements in optimization

methods, ensuring the framework's continuous relevance in the evolving research landscape.

4.5.2 Optimization strategies sets

Three different sets of optimization strategies are supported in our framework, each containing several different combinations of the most promising state-of-the-art optimization techniques that were identified and discussed in Sect. 3, to evaluate the most effective approach to minimize the total energy consumption of an ML model during the training, inference, and loading stages by performing a quantitative analysis of the energy and resource consumption metrics specified in Sect. 4.2.

The first set contains combinations of quantization techniques that can be applied as a post-processing step after training the ML model. The second set contains various methods of compressing the physical representation of an ML model (number of total model parameters or the in-memory size of each parameter), in order to reduce the amount of energy consumed by an ML model during the inference stage and in subsequent retraining that might be necessary due to a change in the underlying data distribution during the operational stage. Finally, the third set contains combinations of the individual techniques included in the other two sets.

The specific combinations of techniques in each set are listed in Table 1. It should be noted that, in order to reduce the computational cost and time spent on the third test set, only the optimal post-training quantization technique according to the results of the first test set in terms of the selected optimization profile is applied to the models of the third set. The reason for this choice is that all the post-training quantization techniques have a negligible computational cost compared to that of the techniques found in the second set. Therefore, by evaluating them beforehand and selecting the optimal one as the one used for the combinations present in the third test set, the total evaluation time is considerably reduced. After preliminary validation, we conclude that this approach does not affect the results obtained with the third set in any meaningful way.

An important detail to note is that the baseline model is trained using a fixed number of epochs rather than using the early stopping technique, as done in Sect. 5.2.1. In particular, 50 epochs are used to train the model by default, although this parameter can be configured by the user. The decision to do so is primarily because the early stopping technique does not always guarantee that the model will complete training at an equal number of epochs each time it is performed. This, in turn, affects the energy consumption metric obtained on each occasion and would not provide a fair metric for comparisons between different optimization techniques.

To apply the post-training quantization techniques, our framework takes advantage of the capabilities provided by TensorFlow Lite, which is a lightweight library for deploying TensorFlow models on resource-constrained devices. More specifically, the TensorFlow Lite converter with the default weight quantization strategy is used to perform the quantization, which aims to reduce latency and model storage size while preserving model accuracy by applying weight quantization per layer. The operations supported by the model and the data types of the input and output layers are explicitly declared during conversion according to the information provided by the TensorFlow Lite documentation [45]. In addition, the user can provide a representative dataset to improve the optimization of the post-training quantization process by allowing biases and activations to be quantized as well. The user must decide on the size of the representative dataset to be provided, which should be large enough to represent the underlying data distribution but small enough to fit in the device memory. A larger dataset will lead to a smaller quantization error, whereas a smaller data set will be faster to process but may lead to a larger quantization error.

For the case of the quantization-aware model fine-tuning technique, our framework makes use of the TensorFlow Model Optimization (TFMOT) library, which is an extension of TensorFlow that contains a set of tools to optimize ML models for deployment and execution. Several quantization strategies are available to the user. The default is the Last Value Quantizer, which aims to preserve model accuracy by quantizing weights according to the range of the last batch of values during training. Although it is worth trying other quantization strategies such as moving average quantizer (which quantizes based on a moving average of values between batches), all values quantizer (which quantizes based on the range of tensor values between all batches), and fixed quantizer (which quantizes based on a fixed range provided as a parameter). As for the tuning parameters used to perform quantization-aware model fine-tuning, the user can either manually optimize these parameters or let the framework through one of the built-in hyperparameter optimization techniques (tree-structured Parzen estimator [TPE] [46], grid search, or random search) to automatically optimize them. In particular, the batch size, the number of epochs to quantify the model after freezing the weights, and the size of the validation split should be optimized. In the case of using a hyperparameter optimization technique, the user must provide the search space for these parameters and the number of maximum evaluations that can be performed during the optimization process.

To select the most suitable combination of techniques to be applied, the hyperparameter optimization technique is guided by the optimization profile defined by the user. In

Table 1 Supported energy efficiency optimization strategies

Set	Opt. Strategy Id.	Opt. Strategy
N/A	0	No optimizations (baseline)
Post-Training Optimization Techniques	1	Full 8-bit Integer (INT8) Weight Quantization
	2	Half-precision Floating-point (FP16) Weight Quantization
	3	Full Integer Weight Quantization with 16-bit Integer (INT16) Activations and 8-bit Integer (INT8) Weights
Training-aware Optimization Techniques	4	Pruning-aware Model Fine-tuning
	5	Quantization-aware Model Fine-tuning
	6	(1) Neural Architecture Search (2) Knowledge Distillation
Combined Optimization Techniques	7	(1) Pruning-aware Model Fine-tuning (2) Quantization-aware Model Fine-tuning
	8	(1) Neural Architecture Search (2) Knowledge Distillation (3) Pruning-aware Model Fine-tuning
	9	(1) Neural Architecture Search (2) Knowledge Distillation (3) Quantization-aware Model Fine-tuning
	10	(1) Neural Architecture Search (2) Knowledge Distillation (3) Pruning-aware Model Fine-tuning (4) Quantization-aware Model Fine-tuning
	11	(1) Pruning-aware Model Fine-tuning (2) Optimal post-training Quantization
	12	(1) Neural Architecture Search (2) Knowledge Distillation (3) Optimal post-training Quantization
	13	(1) Neural Architecture Search (2) Knowledge Distillation (3) Pruning-aware Model Fine-tuning (4) Optimal post-training Quantization

the case of using TPE as the hyperparameter optimization technique, if the performance profile is selected, the target evaluation metric is set as the objective to maximize. In this case, if a combination of values leads to unacceptable performance according to the user-defined threshold, the combination is discarded and another combination is tried instead. On the other hand, if the optimization profile is set to the energy efficiency profile, the TPE technique is used to optimize energy consumption instead of performance. Therefore, if a combination of values leads to an unacceptable energy consumption according to the user-defined profile threshold, the combination is discarded and another combination is tried instead. Furthermore, in case a balanced profile is used as the optimization profile, the objective of the optimization process is set to minimize energy consumption and maximize performance according to user-specified ratios. If, on the other hand, a grid or random search is used, the optimal combination is selected from those tested in the optimization process according to the optimization profile specified by the user. In the event that during hyperparameter optimization a combination of values leads to a performance equal to or better than that of the baseline model, by default, the optimization process will be terminated to save resources and reduce energy consumption. However, this behavior can be disabled if necessary.

To apply weight pruning techniques, the TFMOT library is also used to prune lower-magnitude weights and eliminate redundant information from the model while attempting to maintain its accuracy by keeping only the weights that contribute the most to the model output. The polynomial decay pruning strategy is used to perform the

pruning, which adjusts the degree of sparsity using a polynomial function of the training step. This function is defined in Eq. 1 [47].

$$s_t = s_f + (s_i - s_f)\left(1 - \frac{t - t_0}{n\Delta t}\right)^e \quad \text{for } t \in \{t_0, t_0 + \Delta t, \ldots, \quad t_0 + n\Delta t\} \tag{1}$$

Here s_t is the pruning rate at step t of the training process; s_i is the initial pruning rate; s_f is the final pruning rate; t_0 is the step to start pruning; n is the number of pruning steps; Δt is the rate of pruning steps (pruning frequency); and e is the exponent of the polynomial decay.

The pruning parameters should be optimized to achieve the most appropriate balance between model performance and energy efficiency gain, as in the case of the quantization-aware model fine-tuning technique. Specifically, the initial sparsity that is applied to the model immediately after the start of training and the final sparsity that is the target to be achieved in the last step of the pruning process should be established. Furthermore, pruning is applied every 100 training steps, and the exponent of the polynomial decay function is kept at the default value of 3. We found that the frequency of pruning does not significantly alter the performance of the model or the optimization quality, which is on par with the results of the study presented in [47], and that the default value of the polynomial decay exponent works reasonably well for most cases. However, users are encouraged to experiment with these values to find the most optimal one for their use case.

The fine-tuning hyperparameters, which are the same as the quantization-aware optimization (batch size, number of epochs, and validation split) must also be optimized manually by the user or automatically through the built-in hyperparameter optimization techniques described above. The rest of the hyperparameters (loss function and optimizer) of the training process are kept the same as those used to train the baseline model. For the pruned models, during the conversion to the TensorFlow Lite format, we make use of the experimental sparsity optimization strategy along with the default strategy to take advantage of the sparsity pattern during the conversion to further optimize the latency and storage size of the model.

Finally, in the case of knowledge distillation, we followed the details provided by [41] to implement this technique in our framework. For the distillation process, the same optimizer that is used to train the baseline model is used. The loss function of the student model is also always the same as that used to train the baseline model. The distillation loss function is defined according to Eq. 2 [41].

$$\begin{aligned} L &= \alpha * L_s + (1 - \alpha) * L_d \\ L_d &= T^2 * \mathrm{KLD}(\mathrm{SoftMax}(p/T), \mathrm{SoftMax}(q/T)) \end{aligned} \tag{2}$$

Here α is the hyperparameter that controls the trade-off between the distillation loss and the training loss of the student model. L_s is the training loss of the student model. L_d is the loss function that is being used for knowledge distillation. KLD is the *Kullback Leibler Divergence*. p and q are the outputs of the teacher model and the student model, respectively. T is used as a control parameter that is used to scale the output of the models before applying the SoftMax function to convert them into a softer probability distribution.

The parameters defining the amount of knowledge distillation applied to the student model are controlled by the hyperparameters α and T. These parameters can be either manually adjusted by the user or automatically optimized by the framework to find the values that lead to the best compromise between resource consumption and accuracy loss. As for the training parameters used to train the student model, the batch size and validation split size must be set by the user or by letting the framework automatically find the most optimal values through the built-in hyperparameter optimization techniques described above. Regarding the number of epochs, it is recommended to use the early stopping technique with reasonable patience to avoid overfitting and achieve the best possible performance.

The structure of the student model must be smaller than that of the teacher to provide a positive energy efficiency gain. Therefore, the number of neurons in the hidden layers or the number of hidden layers must be reduced by some factor compared to the teacher model. To do this, the user has three options:

1. Use the NAS technique to automatically determine the most suitable architecture for the student model.
2. Let the user specify the desired reduction factor for the student model and then the framework automatically generates the structure of the student model by modifying the number of neurons in the hidden layers and the number of hidden layers of the teacher model to obtain a model that has a number of parameters that is reduced by the provided value with respect to those of the teacher.
3. Let the user specify the desired structure of the student model.

When using the NAS technique, the best model is selected among the different models generated according to the optimization profile specified by the user.

5 Use case: supervised machine learning for cryptomining detection and mitigation

In this section, we first present the use case selected for the study in order to demonstrate the effectiveness of our proposed framework. Next, we will analyze the base DNN model that will serve as the target for the experimental analysis to be presented later in Sect. 6.

5.1 Use case description

The use case chosen to showcase our methodological framework is based on a realistic deployment of an ML-based cybersecurity solution that demonstrates that novel approaches enabled by ML techniques can allow coping with the development of new cyber threats, such as the detection of malicious encrypted traffic (e.g., cryptomining malware). The setup considered for this use case was deployed in the Mouseworld laboratory, an open laboratory for 5G experimentation located on Telefónica premises [48], and is illustrated in Fig. 2.

The ML-based cybersecurity solution was integrated into TeraFlowSDN, an innovative open-source, cloud-native SDN (software-defined network) controller that offers revolutionary capabilities for both service-level flow management and the integration and management of the underlying network infrastructure, including transport network elements (optical and microwave links), and IP routers, while incorporating cybersecurity capabilities through ML and forensics for multi-tenancy [6].

Since detection and identification of malware network flows traversing the data plane of a network should not be performed on a central ML-based component due to scalability issues and slow response times, we propose to implement a distributed solution where auxiliary ML components are deployed on point of presence (PoP) nodes [7]. To this end, a feature extractor is deployed at the network edge to collect and summarize the packets. The flow statistics aggregated by the feature extractor are sent to an ML classifier. Based on the real-time identification of malicious flows, the ML model will be able to report to the TeraFlowSDN controller at scale to perform a security assessment.

Assuming a typical telecommunication MPLS-based network, a Level 3 VPN service (L3VPN) is deployed using the TeraFlowSDN controller. The controller activates this service using provisioned templates over the standardized IETF NETCONF southbound interface against the different Provider Edge (PE) routers from the manufacturer ADVA. A traffic generator (e.g., IXIA BreakingPoint) emulating the network traffic generated in a branch office, is connected to the leftmost PE to inject network packets representing a mix of normal traffic and cryptomining malware activity. The rightmost PE, the central office, provides internet access offered by the L3VPN service and leveraged by the malware. This specific malicious traffic is represented as a red dashed line in Fig. 2.

As part of the VPN provisioning process, a request for mirroring only the traffic in the logical interfaces that conform to the L3VPN is also included to copy the traffic toward a logical component (distributed attack detector) co-located to the leftmost ADVA router. This logical component (detailed later as distributed attack detector) will extract and calculate statistical features from network flows to be delivered to the TeraFlowSDN controller for further processing by the ML-based Cybersecurity TeraFlowSDN netApp component. This component will identify the attack as a cryptomining activity and propose a mitigation solution to the TeraFlowSDN Core components, which will trigger the corresponding mitigation (e.g., block the malicious connection). This mitigation will be instantiated (green dash-dotted line in Fig. 2) as a new customized access control list (ACL) rule in the ADVA router with specific parameters (protocol TCP, destination IP address, and destination port). It is worth mentioning that Fig. 2, shows an additional branch office in the middle of the figure, to represent a multisite L3VPN functionality where the same rule can be enforced in additional PE

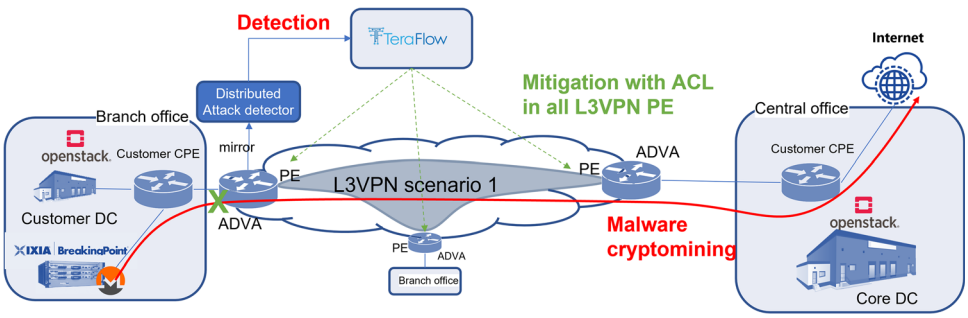

Fig. 2 Global Overview of the Cyberthreat Analysis and Mitigation Use Case

routers, providing protection to all offices of the L3VPN client.

A combination of several Cybersecurity components deployed in the TeraFlowSDN controller focuses on the capture, identification, and mitigation of network threats, implementing a protection layer that is crucial for the correct functionality that SDN controllers need to provide. The Cybersecurity components include two core centralized components, the centralized attack detector, and the attack mitigator, along with a distributed component, the distributed attack detector, that is placed at a remote site.

Among these three components, the one that concerns the work presented in this paper is the centralized attack detector, where the ML model is trained and deployed. The centralized attack detector component provides IP-layer attack detection capabilities and a consolidated attack detection mechanism based on the connection reports from the distributed attack detectors. The centralized attack detector consolidates information received from multiple instances of the distributed attack detector, which allows monitoring of malicious network traffic while forming a view of the security status of IP traffic. From the summarized connection statistics received from the different instances of the distributed attack detector, the centralized attack detector component performs attack detection using an embedded DNN model. The DNN model classifies each connection in the buffer as normal traffic or as part of an attack, and a confidence level decision is derived. From this inference, the centralized attack detector produces a description of the connection, including a confidence value indicating the probability that the connection is an attack or normal traffic. If a connection is detected to be part of an attack with a confidence level at or above a configurable threshold, the centralized attack detector notifies the attack mitigator component with the attack description, providing a full characterization of the attack properties and other relevant information to perform an attack mitigation strategy.

This use case was specifically chosen due to its inherent complexity, demanding highly efficient and accurate DNN models for real-time classification of network traffic. The chosen scenario involves real-time safety-critical operations with limited resources, which demands high-performance models to accurately identify cryptomining activity and minimal energy consumption while ensuring low latency to facilitate swift decision-making, protect the network infrastructure, and ensure the safety of its users. These stringent requirements are accentuated by the need to quickly analyze large volumes of network traffic with minimal delay at several remote locations at the network edge, which adds up to an immensely high rate of inference processes per second. Consequently, energy efficiency becomes a crucial factor for practical implementation, highlighting the important real-world challenge that our framework aims to address.

An in-depth analysis of the cryptomining connection detection methodology using ML and DL models is provided in [5]. After a thorough comparison including fully connected neural networks (FCNNs), random forest, classification and regression trees, and C4.5, this study identified FCNNs as one of the best-performing models to recognize patterns in network traffic that may indicate the presence of cryptomining malware. In light of the results obtained in this exhaustive comparative study, the centralized attack detector currently employs a DNN model based on a multi-layer FCNN as its classifier.

It should be noted that, although in our experimental evaluation we focus on a specific DNN architecture (FCNNs) and a specific use case (cryptocurrency mining detection), the concepts and principles presented in this article and on which we have relied to design our framework are applicable to any type of DNN architecture and any type of application. The reason we chose an FCNN-based model as the target for optimization and a cryptocurrency mining detection use case is because: 1) FCNNs are the most versatile of all neural network architectures and can be applied to any type of application, including image recognition and natural language processing; 2) cryptocurrency mining detection is a complex application that requires very efficient and high-performance DNN models for real-time classification of network traffic with high accuracy due to the critical nature of the application, which makes it restrictive in terms of energy consumption and resource utilization; and 3) the data and features used in the cryptocurrency mining detection application have no underlying topology, which makes them well suited to be analyzed using FCNNs. In contrast, if the data had a spatial or temporal topology, the architecture of choice would have been a convolutional neural network or a recurrent neural network.

5.2 Analysis of the baseline deep neural network model

In this subsection, we describe in detail the model we used as the basis for our energy efficiency analysis. First, we describe the setup we used to collect the data used to train the model. Next, we present the structure of the model and the procedure that was followed to train it. Finally, we evaluate the model using several standard performance metrics.

5.2.1 Training of the baseline deep neural network model

The dataset that was used to train the baseline DNN model that will serve as the target in the demonstration of the

proposed methodology has been developed for the precise task of detecting cryptomining attacks [5]. This dataset was generated in the Mouseworld lab [48] in Telefónica I+D premises. This emulation environment allows configuring and executing specific attacks mixed with normal traffic (e.g., web, file hosting, streaming, etc.) instantiating virtual machines that deploy normal traffic and specific attack clients connected to real servers located at different points on the Internet. In this way, the Mouseworld Lab can be used to set up and emulate attack scenarios in a controlled way and to generate and collect in PCAP files all packets of the attack and normal traffic to be used later for the training and testing of ML models. One key feature of the Mouseworld Lab is the repeatability capacity, which allows us to evaluate different mitigation tools or versions in the same conditions and using similar statistical patterns.

The data collected for our study in the Mouseworld Lab contain traffic samples represented by a set of flow (TCP connection) statistics derived from network packets using the Tstat tool. The statistics of a TCP connection were calculated periodically (at fixed intervals or when a new burst of packets was received), resulting in many different examples in the dataset for the same flow representing the state of the connection over its lifetime. These traffic data were labeled to create the dataset that was used to train and test the cryptomining detector. In particular, two types of traffic can be found in the dataset: samples corresponding to normal traffic and samples corresponding to cryptomining attacks. In this case, each sample of the dataset was tagged as either 0 (normal traffic) or 1 (cryptomining attack traffic) using the IPs and ports of the known attack connections. Once labeled, the dataset was randomly partitioned into two subsets: the training set, utilized for model learning, and the test set, reserved for evaluating the final model's performance. The proportion of data assigned to each subset follows a 50:50 ratio. From the training split, 20% of the data was reserved for the validation set, employed for hyperparameter fine-tuning and model selection.

The distribution of data samples across different sets used for training, validating, and testing the model is provided below.

- *Training dataset size*: 83,591 data samples
- *Validation dataset size*: 20,898 data samples
- *Test dataset size*: 97,558 data samples
- Total: 202,047 data samples

Therefore, we used the TensorFlow library to train the FCNN classifier to predict whether a connection corresponds to cryptomining activity or not according to all features derived from Tstat statistics except IPs and ports that we used only to label the dataset (class labels). Note that source and destination IPs and ports can easily be changed by the attacker and, therefore, do not provide significant information to the ML-based detector.

The structure of the FCNN model that was used as a baseline model is specified below. In particular, the model consists of a stack of three fully connected layers with 20, 30, and 10 neurons with ReLU activation followed by a fully connected layer with two neurons and SoftMax activation as the output layer. The training hyperparameters are as follows. We use a batch size of 4096 and the Adam optimizer with a learning rate of 0.001. Furthermore, we use the early stopping technique to automatically terminate the training process if the validation loss does not improve for 20 epochs, restoring the model weights to those obtained in the epoch with the lowest validation loss after training is complete. Finally, as a loss function, we use the categorical cross-entropy function.

Although the accuracy of the model using all these features is already high, it was observed that many of them do not contribute significantly to the prediction performance and can be ignored to improve the training efficiency and model inference. Therefore, we decided to make a random selection of the most commonly used features and managed to reduce the required input to ten features, while the F1-score was still high ($> 95\%$). We list the features that we selected in Table 2. Note that if a feature has a CS (Client–Server) and SC (Server-Client) identifier, it is because it has been measured in both directions. However, if a feature has only one identifier, it is because it has been measured in the direction indicated by the identifier type (CS or SC).

An extended description of the features employed and their respective purposes within the TCP protocol is provided below.

- *SYN count* Number of synchronization (SYN) segments observed, including retransmissions. SYN segments are part of the TCP three-way handshake used to establish a connection between two devices.
- *Window Scale* Scaling values negotiated during TCP connection establishment. The scale factor adjusts the size of the TCP window, allowing for more precise flow control.
- *MSS* Maximum Segment Size (MSS) declared during TCP connection setup. MSS specifies the maximum amount of data that can be sent in a single TCP segment.
- *Max Seg Size* Maximum segment size observed during communication. It represents the largest amount of data carried in a single TCP segment.
- *Min Seg Size* Minimum segment size observed during communication. It indicates the smallest amount of data carried in a single TCP segment.

Table 2 Selected features of the Crypto dataset to train the baseline model

CS ID	SC ID	Name	Type	Description
13	27	SYN count	Numeric	Number of SYN segments observed (including rtx)
–	90	Window scale	–	Scaling values negotiated [scale factor]
70	–	MSS	Bytes	MSS declared
71	94	Max seg size	Bytes	Maximum segment size observed
72	95	Min seg size	Bytes	Minimum segment size observed
73	96	Win max	Bytes	Maximum receiver window announced (already scaled by the window scale factor)
74	97	Win min	Bytes	Minimum receiver window announced (already scaled by the window scale factor)
76	99	cwin max	Bytes	Maximum in-flight-size computed as the difference between the largest sequence number so far, and the corresponding last ACK message on the reverse path. It is an estimate of the congestion window
77	100	cwin min	Bytes	Minimum in-flight-size
78	–	Initial cwin	Bytes	First in-flight size, or total number of unack-ed bytes sent before receiving the first ACK segment

CS Client to server traffic

SC Server to client traffic

- *Win Max* Maximum receiver window announced during TCP communication. The receiver window is a flow control mechanism, and this value is scaled by the negotiated window scale factor.
- *Win Min* Minimum receiver window announced during TCP communication. Similar to the maximum receiver window, this value is scaled by the window scale factor.
- *Cwin Max* Maximum in-flight-size computed as the difference between the largest sequence number and the corresponding last ACK message on the reverse path. It serves as an estimate of the congestion window in TCP.
- *Cwin Min* Minimum in-flight-size observed during TCP communication. It represents the minimum amount of unacknowledged bytes sent.
- *Initial Cwin* Initial in-flight size, indicating the total number of unacknowledged bytes sent before receiving the first ACK segment during TCP connection establishment.

All data were standardized to ensure that the mean of the samples of each feature was 0 and the standard deviation was 1. This was done so that the scale of each variable did not cause one variable to dominate the results. We found that standardization significantly improved the results.

5.2.2 Performance evaluation of the baseline deep neural network model

We used three well-known metrics (accuracy, balanced accuracy, and F1-score) to validate the performance of the non-optimized DNN model in a reserved portion of the data that was not used during training and that represents 20% of the total dataset. We incorporated balanced accuracy among the evaluation metrics to account for the imbalances that exist in the dataset.

- Accuracy: 100%
- Balanced Accuracy: 99.5%
- F1-score: 1

We will use these results as a basis for comparison when evaluating the performance of the models obtained with the different optimization techniques that will be tested in the following sections.

6 Experimental evaluation

This section presents the empirical findings of the validation of the open source framework proposed in this work, demonstrating its usefulness as an experimental platform for optimizing in an automated way the energy efficiency of DNNs. To demonstrate the effectiveness of the framework, we optimize a DNN classifier deployed in a cybersecurity component of an SDN controller, in which the cybersecurity component is responsible for monitoring and detecting cryptomining activity in the network. To showcase the capabilities of the framework, the results obtained at the end of the optimization process are presented for the three available optimization profiles.

We begin by detailing in Sect. 6.1 the experimental setup, which includes a description of the main parameters configured in the framework for the optimization process and the hardware platform used to run the framework. In addition, in Sect. 6.2, we detail the framework's ability to perform an automatic hyperparameter selection process, with the goal of achieving optimal results with each supported technique. Subsequently, Sect. 6.3 examines the experimental results acquired during the model inference phase for all applied optimization strategies. Finally, in Sect. 6.4, we complement Sect. 6.2 results with a detailed analysis of the experimental results obtained in the training and optimization of the target model.

6.1 Framework setup and environmental configuration

To demonstrate the effectiveness of the proposed methodology, we have performed an experimental evaluation in which we have applied the proposed methodology with all combinations of supported techniques to the cryptomining detector described in Sect. 5.2 using the default parameters and settings of our framework (number of repetitions set to 5 and sample measurement time interval of 1 s). In this way, we were able to obtain a detailed analysis of the trade-off between energy and accuracy that exists when energy optimization techniques are applied to an FCNN.

To carry out the optimization process with the framework, we will use the three available profiles we proposed (energy efficiency profile, performance profile, and balanced profile) to select the most appropriate optimization strategy. We set as the performance threshold a minimum acceptable reduction in energy consumption with respect to the non-optimized model of 25% and a minimum balanced accuracy of 0.9. Furthermore, to apply the balanced profile, we set the ratio of these two factors as 0.5 for both in order to obtain the optimization strategy that leads to the most balanced results between the two objectives and analyze its comparison with those obtained with the other two profiles.

The reason why we use balanced accuracy as the target performance metric of the optimization process is that, in our case, the number of positive and negative samples is highly imbalanced (positive samples are much less frequent than negative samples). This means that using the accuracy metric as the optimization objective would lead to the selection of a model configuration with very low accuracy on the positive samples. This is not the case for the balanced accuracy metric, as this metric can be interpreted as a generalization of accuracy that takes into account the class imbalance, improving the fairness of the optimization process. It is important to note that the value obtained with balanced accuracy is always lower than or equal to the accuracy, and therefore we can use it as a lower bound to evaluate the performance of different optimization strategies. It should be noted that, although we have used the balanced accuracy metric to drive the optimization process, the comparative analysis of the results also includes the accuracy and F1-score metrics. We will only use balanced accuracy as the target performance metric for our optimization process because, as explained above, it is a more restrictive and more reliable metric to evaluate the effectiveness of the different optimization strategies that have been applied. Furthermore, we use a balanced accuracy of 0.9 as a threshold because it provides an acceptable accuracy in our case while maintaining adequate energy efficiency.

On the other hand, we have chosen to focus our evaluation on CPU-based systems, as they are the most widely used for the deployment of DNN-based applications due to their lower cost, scalability, and flexibility. However, the fundamental principles and concepts presented in this article are applicable to any type of hardware platform, including GPUs and custom DNN accelerators.

The hardware platform used to validate the proposed methodology is a system with an Intel(R) Core(TM) i7-2600 CPU (Sandy Bridge microarchitecture; base clock 3.40 GHz; turbo boost 3.80GHz; and 8 MB cache). The system has 32 GB of RAM and Ubuntu 20.04.5 (with 5.15.0-48-generic Linux kernel) was used as the operating system. As the run-time, we used Python (version 3.10.6) and the following packages: TensorFlow (version 2.9.2), TensorFlow Model Optimization (version 0.7.3), psutil (version 5.9.4), and powerstat (version 0.02.27).

6.2 Selection of hyperparameters used for the application of optimization techniques

To determine the hyperparameters of the optimization techniques, a fully automated procedure was used. The chosen technique involved using a grid search approach, where predefined search spaces were utilized for each parameter. These search spaces were considered reasonable based on the literature review and can be considered suitable default options. Therefore, the optimization techniques were applied using the most optimal values of the hyperparameters, which were automatically determined by this optimization process. More specifically, the optimal hyperparameters were automatically identified by the framework using a grid search procedure with cross-validation and a three-fold configuration. The goal was to identify the optimal combination of hyperparameters that would achieve the best balance between model performance and efficiency (balanced profile), following the framework parameters described in Sect. 6.1. Note,

however, that some parameters cannot be optimized automatically, as they require domain knowledge and prior experience (e.g., learning rate) or usage-specific considerations (e.g., batch size). In these cases, we use values that are commonly recommended in the literature as good starting points and therefore do not require extensive tuning.

In order to determine the optimal representative dataset size for the application of post-training weight quantization techniques, the framework performs a cross-validation procedure, using three folds and evaluating a range of representative dataset sizes from 0.1 to 1.0. This representative dataset allows calibration of the range of variable floating-point tensors in the model (e.g., activations, model input, and output), ensuring optimal inference accuracy for post-training weight quantization. In our particular case, a 25% representative dataset was selected to improve the optimization of the post-training weight quantization process. Note that a representative dataset is not required for half-precision floating-point weight quantization technique and, therefore, was only used for the other two post-training weight quantization techniques.

In the case of the quantization-aware model fine-tuning technique, we adopt the quantization of 8-bit weights per layer using the Last Value Quantizer strategy, which ensures that the model accuracy is maintained during the quantization process. The parameter grid used for this purpose was the one pre-configured in the framework, consisting of batch sizes of 256, 512, 1024, and 2048, and the number of epochs set to 10, 20, and 30. After experimenting with different values for the batch size and number of epochs to quantify the model after freezing the weights, our framework identified the optimal values of the fitting parameters for a batch size of 256 and a number of epochs of 10. These values provide the best trade-off between accuracy loss and energy consumption in our case. It should be noted that the rest of the hyperparameters of the training process, such as the loss function and the optimizer, were kept the same as those used to train the reference model.

Two pruning techniques, constant sparsity and polynomial decay sparsity, were also tested. The results demonstrated that the polynomial decay sparsity technique yielded the best performance, and thus, it was selected for further analysis. The hyperparameters for this technique included the initial sparsity, final sparsity, number of epochs, and batch size used for fine-tuning. The grid search was conducted with the predetermined parameter grid configured in the framework for this technique, including four batch sizes (256, 512, 1024, and 2048), three epoch values (10, 20, and 30), four initial sparsity values (0.50, 0.60, 0.70, and 0.80), and four final sparsity values (0.60, 0.70, 0.80, and 0.90). Based on the optimal hyperparameter values, weight pruning was applied every 100 training steps during the training process using a polynomial decay function with an exponent of 3. The initial sparsity was set to 70%, and the final sparsity was set to 90%. For fine-tuning, a batch size of 1024 and 10 epochs were determined to be the optimal values. The loss function and optimizer used during training were the same as those used to train the baseline model.

Finally, in the case of knowledge distillation, the structure of the student model was obtained by searching the neural architecture with the balanced optimization profile to obtain the model architecture that achieves the best compromise between reduced energy consumption and performance. For this purpose, a grid search was used to define the structure of the student model. The training hyperparameters used in the search procedure were kept as the defaults used for knowledge distillation (i.e., the same as the baseline model). The space of architectures explored includes two- to three-layer architectures, with the number of neurons per layer ranging from 4 to 64, in steps of powers of two. The activation function used in all layers was ReLU and the output layer had two neurons with SoftMax activation as the output layer. The student model that was obtained consists of a fully connected two-layer stack with 8 and 4 neurons with ReLU activation followed by a fully connected layer with two neurons and SoftMax activation as the output layer. Note that the student model is more than 100 times smaller than the teacher in terms of the number of parameters.

In relation to the parameters that determine the degree of knowledge distillation applied to the student model, two parameters were defined, namely α and T. The former parameter, α, is responsible for controlling the amount of knowledge transfer between the teacher and student models by weighing the loss function defined in Eq. 2. The latter parameter, T, controls the sharpness of the SoftMax function. The parameter grid search space used is the one pre-configured in the framework and is defined by a range of values for α and T, specifically [0.1, 0.2, 0.3, 0.4, 0.5] and [0.5, 1, 1.5, 2, 2.5, 3], respectively. The framework automatically determined the best parameters based on the results of the validation set. In our case, the optimal values were found to be α=0.1 and T=2, which provided the best trade-off between model performance and power consumption. These values were then used to train the student model. It should be noted that the training parameters, such as batch size and validation split size, remained the same as those used to train the baseline model in order to ensure consistency between the baseline model and the student model training process and thus allow for a fair comparison. In addition, the student model was trained using the early stopping technique until no improvement in the validation set was observed for an unlimited number of epochs

to avoid overfitting and achieve the best possible performance.

6.3 Energy cost analysis in the inference stage

In this section, we analyze the results obtained in the inference stage of the optimized models. These results determine the optimization strategy that provides (i) the best results in terms of energy consumption and (ii) the best compromise between energy efficiency and performance according to the three optimization profiles defined in Sect. 4.3. Three different batch sizes (small: 32, medium: 256, and large: 1024) were tested to analyze the influence of the batch size used to perform the prediction on the results obtained. The results obtained for the optimization strategies that have been applied to the objective model are shown in Tables 3, 4 and 5 for the three batch sizes considered, respectively. The results shown in these three tables correspond to the mean, standard deviation, and maximum energy consumption and resource utilization metrics derived from the aggregation of the measured values collected over the duration of the model inference at 1-second intervals and over 5 iterations for each optimization strategy.

The first thing we observe from the results presented in Tables 3, 4 and 5 is that almost all optimization strategies lead to a significant reduction in energy consumption, exceeding in most cases the threshold of reduction in energy consumption compared to the non-optimized model that was set at the beginning of the experimental evaluation.

In general, we observe that the optimization strategy based on the application of the neural architecture search and knowledge distillation techniques is the one that provides the largest reduction in energy consumption in all cases studied. However, in some cases, the optimization strategy based on the application of the neural architecture search and knowledge distillation followed by the pruning-aware fine-tuning technique and the quantization-aware model fine-tuning provides a slight gain over the former but with a much greater performance penalty. In addition, the optimization strategy that provides the best energy efficiency gain with no performance degradation is the half-precision floating-point weight quantization technique.

The reason why knowledge distillation is the optimization technique that provides the greatest energy reduction, in this case, is that this technique provides the most drastic reduction in the number of parameters of the objective model, which directly translates into a reduction in the amount of energy required to perform the inference stage of the ML process. However, it is important to note that the model that was used as the student model for the application of the knowledge distillation technique would not be able to provide accurate predictions if used by itself as the target model, as it lacks the complexity necessary to correctly learn the relationship between the input and output variables. We found that the student model provided an accuracy of 0.995, a balanced accuracy of 0.5, and an F1-score of 0.0 on several tests. These results clearly indicate that this model, when trained without knowledge distillation, is only able to return negative predictions in all cases (normal traffic), as it is not able to recognize patterns that distinguish positive samples (cryptomining traffic) from negative ones. Therefore, the application of knowledge distillation by transferring the knowledge learned by the teacher model (which is able to successfully learn to separate the positive from the negative class) to a smaller model is essential to reduce the model size to a high degree and maintain acceptable accuracy.

In the next Sects. 6.3.1, 6.3.2, and 6.3.3, we analyze the results segregated by batch size. Finally, regarding the use of computational resources (e.g., CPU, memory, and disk) influence directly in the energy consumption of the system, we provide in Sect. 9 a detailed analysis of the use of computing resources (CPU, memory, and disk), taking into account the optimization strategies and profiles mentioned above, as well as batch sizes.

6.3.1 Small batch size (32)

As presented in Table 3 and summarized in Fig. 3, in the case of using a small batch size of 32, globally taking into account all optimization strategies, the reduction in energy consumption obtained with the application of the optimization strategy consisting of the application of the neural architecture search and knowledge distillation technique followed by a half-precision floating-point weight quantization (which turned out to be the optimal post-training quantization for this particular case, as can be seen in Table 3) is the most favorable, providing an 80.741% reduction in total average energy consumption relative to the baseline model. The second optimization strategy that provided the largest reduction in energy consumption was achieved with the application of the neural architecture search and knowledge distillation without any subsequent post-training quantization technique, leading to a reduction of 79.927%. In the third place, we found the optimization strategy based on the application of the neural architecture search and the knowledge distillation technique followed by the pruning-aware model fine-tuning technique as the second step, which provides a reduction in energy consumption of 55.427%.

Table 3 Average, standard deviation and maximum energy consumption and resource utilization metrics derived from the aggregation of measured values collected over the duration of model inference at 1-second intervals and over 5 iterations for each optimization strategy using a batch size of 32 to perform the prediction

Opt. Strategy Id.	0	1	2	3	4	5	6[a,c]	7	8[b]	9	10	11	12	13
Total Avg. CPU Energy Consumption (J)	4.127	3.691	1.847	8.418	1.879	3.696	0.828	3.921	1.84	3.882	3.872	1.905	***0.795***	1.858
Percentage of Total Avg. CPU Energy Consumption Reduction (%)	N/A	10.576	55.243	−103.962	54.482	10.456	79.927	4.999	55.427	5.944	6.187	53.832	***80.741***	54.974
Avg. CPU Power Draw (W)	***15.504***	15.77	15.757	15.668	15.807	15.67	15.731	16.018	15.862	16.101	16.064	16.093	15.902	15.883
Std. Dev. CPU Power Draw (W)	0.083	0.349	0.279	0.147	0.281	0.259	0.12	0.375	0.34	0.302	0.29	0.319	***0.076***	0.216
Max. CPU Power Draw (W)	***25.69***	26.82	26.82	26	26.96	26.66	26.03	27.23	27.46	27.18	27.3	27.44	26.38	26.55
Avg. CPU Usage (%)	6.244	6.269	6.244	6.244	6.269	6.244	6.244	6.269	6.244	***6.219***	6.269	6.244	6.244	6.269
Std. Dev. CPU Usage (%)	***0***	0.05	0.112	0.079	0.05	***0***	***0***	0.05	***0***	0.05	0.094	***0***	0.079	0.094
Max. CPU Usage (%)	***12.488***	12.738	12.738	12.738	12.738	***12.488***	***12.488***	12.738	***12.488***	***12.488***	12.738	***12.488***	12.738	12.738
Avg. CPU Frequency (MHz)	1.913	2.002	1.936	2.062	1.899	1.985	1.948	2.002	***1.869***	1.952	2.106	1.966	1.99	1.959
Std. Dev. CPU Frequency (MHz)	0.072	0.07	0.087	0.107	0.039	0.08	0.088	0.095	***0.008***	0.052	0.195	0.078	0.143	0.109
Max. CPU Frequency (MHz)	2.138	2.149	2.16	2.406	2.137	2.27	2.318	2.163	***2.136***	2.145	2.456	2.14	2.408	2.177
Avg. RAM Memory USS (B)	10.105	**6.902**	9.97	9.85	9.985	9.876	9.869	9.89	10.02	9.881	9.871	10.023	9.894	10.029
Std. Dev. RAM Memory USS (B)	0.01	0.021	0.012	0.025	0.012	0.014	0.014	0.013	0.022	0.011	0.016	0.023	0.022	***0.009***
Max. RAM Memory USS (B)	16.75	***10.445***	16.473	16.344	16.539	16.34	16.289	16.352	16.598	16.352	16.367	16.617	16.367	16.605
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

[a]Best optimization strategy according to the energy efficiency profile

[b]Best optimization strategy according to the performance profile

[c]Best optimization strategy according to the balanced profile

Underlined values: Top three optimization strategies that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

J: Joules. W: Watts. B: Bytes. MHz: Megahertz

Table 4 Average, standard deviation, and maximum energy consumption and resource utilization metrics derived from the aggregation of measured values collected over the duration of model inference at 1-second intervals and over 5 iterations for each optimization strategy using a batch size of 256 to perform the prediction

Opt. Strategy Id.	0	1	2[b]	3	4	5	6[a,c]	7	8	9	10	11	12	13
Total Avg. CPU Energy Consumption (J)	4.309	3.651	1.854	8.495	1.887	3.732	***0.763***	3.682	1.867	3.748	3.827	1.931	0.817	1.873
Percentage of Total Avg. CPU Energy Consumption Reduction (%)	N/A	15.28	56.975	−97.138	56.199	13.398	***82.304***	14.551	56.666	13.025	11.195	55.188	81.046	56.532
Avg. CPU Power Draw (W)	16.138	15.794	***15.691***	15.773	15.782	15.758	15.806	15.88	15.898	16.088	15.961	16.11	15.842	15.98
Std. Dev. CPU Power Draw (W)	0.851	0.267	0.119	0.055	0.289	0.273	0.263	0.232	0.28	0.371	0.383	0.278	0.074	***0.054***
Max. CPU Power Draw (W)	25.86	26.94	***25.81***	26.05	26.89	26.9	27.03	26.69	27.21	27.75	27.23	27.26	26.22	26.49
Avg. CPU Usage (%)	6.219	***6.169***	6.194	6.269	6.194	6.244	6.244	6.219	6.269	6.244	6.219	6.244	6.294	6.219
Std. Dev. CPU Usage (%)	0.094	0.061	0.061	0.05	0.061	***0***	***0***	0.05	0.094	***0***	0.094	0.079	0.1	0.094
Max. CPU Usage (%)	12.738	***12.488***	***12.488***	12.738	***12.488***	***12.488***	***12.488***	***12.488***	12.738	***12.488***	12.738	12.738	12.988	12.738
Avg. CPU Frequency (MHz)	1.975	1.976	2.012	2.195	***1.89***	1.991	1.935	1.977	1.952	2.021	1.942	1.969	1.92	1.918
Std. Dev. CPU Frequency (MHz)	0.089	0.101	0.13	0.236	***0.047***	0.135	0.091	0.153	0.101	0.165	0.075	0.086	0.084	0.071
Max. CPU Frequency (MHz)	2.156	2.404	2.416	3.168	2.145	2.252	2.414	2.413	2.164	2.412	2.146	2.17	***2.141***	2.15
Avg. RAM Memory USS (B)	10.559	***7.415***	10.509	10.393	10.496	10.423	10.356	10.427	10.545	10.422	10.39	10.552	10.386	10.543
Std. Dev. RAM Memory USS (B)	0.017	0.031	0.018	0.021	0.025	0.024	0.014	0.016	***0.013***	0.024	0.016	0.029	0.021	0.027
Max. RAM Memory USS (B)	17.387	***11.215***	17.289	17.16	17.332	17.168	17.051	17.16	17.324	17.16	17.078	17.375	17.105	17.336
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

[a]Best optimization strategy according to the energy efficiency profile

[b]Best optimization strategy according to the performance profile

[c]Best optimization strategy according to the balanced profile

Underlined values: Top three optimization strategies that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

J: Joules. W: Watts. B: Bytes. MHz: Megahertz

Table 5 Average, standard deviation, and maximum energy consumption and resource utilization metrics derived from the aggregation of measured values collected over the duration of model inference at 1-second intervals and over 5 iterations for each optimization strategy using a batch size of 1024 to perform the prediction

Opt. Strategy Id.	0	1	2[b]	3	4	5	6[a,c]	7	8	9	10	11	12	13
Total Avg. CPU Energy Consumption (J)	4.463	3.604	1.865	8.498	1.984	3.791	0.798	3.766	1.871	3.783	3.868	1.898	***0.795***	1.9
Percentage of Total Avg. CPU Energy Consumption Reduction (%)	N/A	19.249	58.208	−90.411	55.555	15.05	82.124	15.609	58.074	15.242	13.322	57.47	***82.183***	57.437
Avg. CPU Power Draw (W)	16.764	***15.613***	15.689	15.763	16.876	15.767	15.753	15.967	15.848	15.958	16.323	16.521	16.839	16.029
Std. Dev. CPU Power Draw (W)	2.774	0.283	0.337	***0.213***	1.774	0.421	0.448	0.818	0.312	0.82	1.003	2.381	2.789	0.729
Max. CPU Power Draw (W)	38.96	26.71	26.8	25.92	25.84	***25.66***	26.46	26.85	27.19	26.09	27.07	36.97	26.22	26.48
Avg. CPU Usage (%)	6.244	6.219	6.269	6.244	6.244	6.244	6.244	6.294	6.219	6.219	6.219	6.219	6.244	***6.194***
Std. Dev. CPU Usage (%)	***0***	0.094	0.05	0.079	0.079	***0***	0.079	0.061	0.05	0.05	0.05	0.05	0.079	0.061
Max. CPU Usage (%)	*textub* 12.488	12.738	12.738	12.738	12.738	***12.488***	12.738	12.738	***12.488***	***12.488***	***12.488***	***12.488***	12.738	***12.488***
Avg. CPU Frequency (MHz)	2.298	1.947	1.922	2.051	1.938	1.951	***1.891***	1.961	1.914	1.906	1.939	1.922	1.974	1.925
Std. Dev. CPU Frequency (MHz)	0.357	0.095	0.074	0.053	0.086	0.075	***0.035***	0.113	0.068	0.069	0.133	0.089	0.143	0.089
Max. CPU Frequency (MHz)	3.54	2.14	2.14	***2.139***	2.151	2.151	2.152	2.344	2.145	2.294	2.411	2.342	2.324	2.159
Avg. RAM Memory USS (B)	10.195	***6.822***	9.973	9.854	9.953	9.846	9.875	9.875	10.021	9.9	9.91	10.07	9.911	10.075
Std. Dev. RAM Memory USS (B)	0.02	0.018	0.02	0.027	0.013	0.029	0.013	0.014	0.02	0.021	0.013	0.012	***0.011***	0.018
Max. RAM Memory USS (B)	16.824	***10.348***	16.469	16.332	16.492	16.348	16.285	16.328	16.586	16.375	16.387	16.609	16.344	16.617
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

[a]Best optimization strategy according to the energy efficiency profile

[b]Best optimization strategy according to the performance profile

[c]Best optimization strategy according to the balanced profile

Underlined values: Top three optimization strategies that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

J: Joules. W: Watts. B: Bytes. MHz: Megahertz

The best optimization strategy according to the energy efficiency profile and the balanced profile is the one based on the application of the neural architecture search and knowledge distillation. The results show that this optimization strategy provides a very favorable balance between energy and accuracy, achieving the highest reduction in energy consumption with minimal degradation in model performance. This optimization strategy provides energy savings of 80.741% compared to the baseline model and performance that meets the minimum performance threshold established in the optimization process (balanced accuracy of 0.9), achieving an accuracy of 0.984, a balanced accuracy of 0.987, and an F1-score of 0.989. As can be seen, some degradation in model performance is obtained with these optimization strategies. However, the performance degradation is negligible (0.08% loss in the balanced accuracy, 0.016% loss in the accuracy, and 0.11 loss in the F1-score) and exceeds the minimum performance threshold established for the optimization process. In this particular case, this small degradation is justified by the significant reduction in energy consumption achieved with this optimization strategy.

On the other hand, the optimization strategy that provides the best performance when the small batch size is used for prediction is the one based on the application of the neural architecture search and knowledge distillation technique and a pruning-aware model fine-tuning applied subsequently. As can be seen from the results presented in Table 3, this optimization strategy provides a model with an accuracy of 1, a balanced accuracy of 0.995, and an F1-score of 1, which is the same performance as that exhibited by the baseline model. In addition, the optimized model obtained with this optimization strategy provides a reduction that exceeds the predefined threshold of 25% reduction in energy consumption compared to the baseline model, achieving a reduction in energy consumption of 55.42%, which is more than double the minimum desired efficiency gain established as the threshold for the optimization process. Therefore, the results obtained demonstrate that it is possible to obtain a model with the same performance as the baseline model by reducing its energy consumption by more than half, thus achieving an optimal balance between energy and accuracy for situations where performance degradation is a concern.

6.3.2 Medium batch size (256)

In the results shown in Table 4 and summarized in Fig. 4, we observe that, if the batch size used for the prediction is increased to 256, the optimization strategy that provides the highest reduction in energy consumption with respect to the baseline model is the one based on the application of the neural architecture search and knowledge distillation technique, providing a reduction of 82.304% in energy consumption. Second, we find the optimization strategy consisting of the application of the neural architecture search and knowledge distillation technique followed by a pruning-aware model fine-tuning, providing a reduction in energy consumption of 81.046%. The optimization strategy based on the application of the half-precision floating-point weight quantization is in third place, providing a reduction in energy consumption of 56.975%.

The results shown in Table 4 are closely aligned with those obtained with the small batch size of 32, since in this case the optimization strategy selected by the energy efficiency and balanced profiles is also the one based on neural architecture search and knowledge distillation without any post-training quantization technique. This optimization strategy provides an energy reduction of 82.304% and shows a performance that far exceeds the minimum performance threshold established in the optimization process (balanced accuracy of 0.9), achieving an accuracy of 0.984, a balanced accuracy of 0.987 and an F1-score of 0.989. As can be seen from the results presented in Table 4, the loss of performance with respect to the baseline model is minimal (0.08% for the balanced accuracy, 0.016% for the accuracy, and 0.11 for the F1-score). Therefore, this optimization strategy provides a very favorable balance between energy and performance, achieving a very high reduction in energy consumption with a negligible loss in performance.

Regarding the optimization strategy that is selected by the performance optimization profile when the batch size of 256 is used for inference, the results show that the strategy based on the application of half-precision floating-point weight quantization provides a model that shows absolutely no degradation in performance with respect to the baseline model, exhibiting an accuracy of 1, a balanced accuracy of 0.995 and an F1-score of 1. Furthermore, the energy reduction achieved is 56.975%, which is more than double the 25% reduction that was established as a threshold for the optimization process. Therefore, we can conclude that the optimization strategy based on the application of half-precision floating-point weight quantization provides a model with the best energy efficiency that can be obtained when performance degradation is not tolerated by the application requirements.

6.3.3 Large batch size (1024)

The results shown in Table 5 and summarized in Fig. 5 corresponding to the case in which the batch size is increased to 1024 show some similarities with those obtained for a batch size of 256. In this case, the optimization strategy based on the application of the neural architecture search and knowledge distillation technique

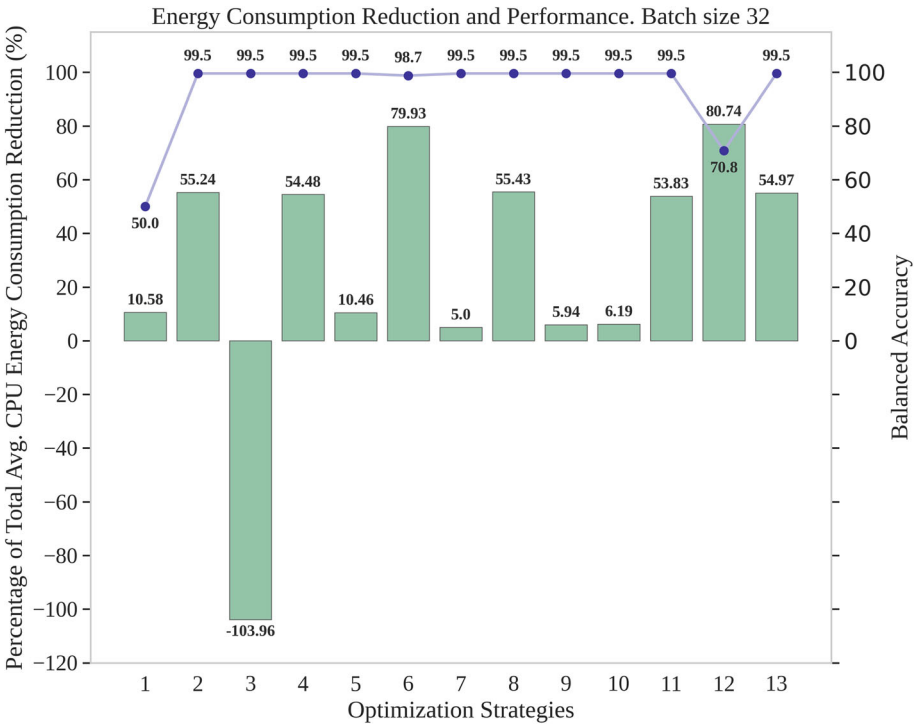

Fig. 3 Comparative analysis showing the energy efficiency and performance metrics of all optimization strategies evaluated with a small batch of 32 samples

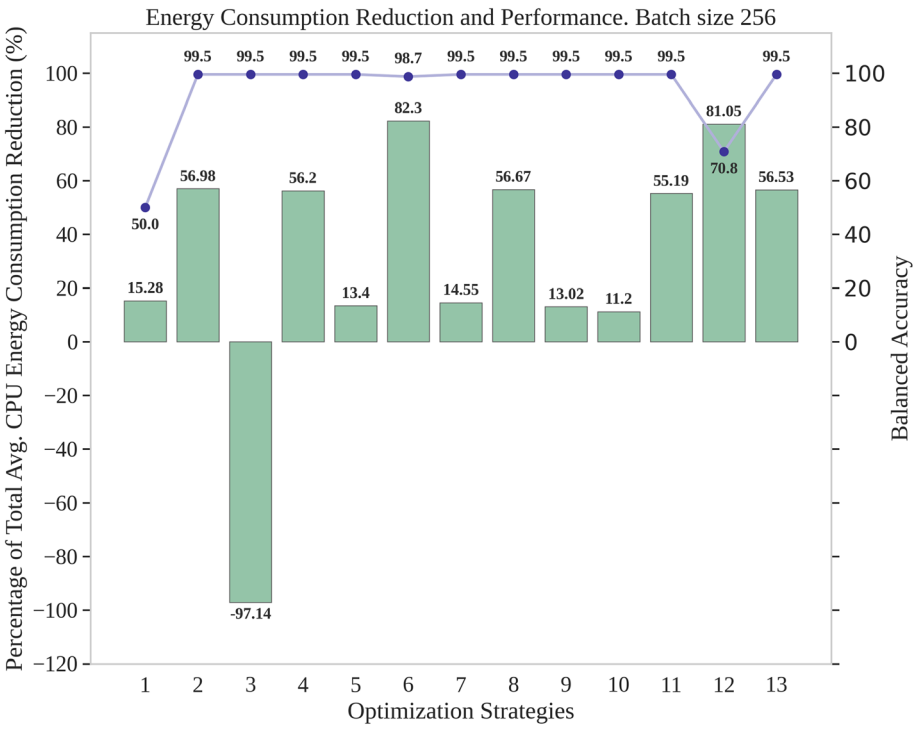

Fig. 4 Comparative analysis showing the energy efficiency and performance metrics of all optimization strategies evaluated with a medium batch of 256 samples

followed by a pruning-aware model fine-tuning is the best option in terms of reducing energy consumption, providing 82.183% energy savings. The second best result is the optimization strategy consisting of the application of the neural architecture search and knowledge distillation technique, which provides a reduction in energy consumption of 82.124%. In third place is the optimization strategy based on half-precision floating-point weight quantization, which reduces energy consumption by 58.208%.

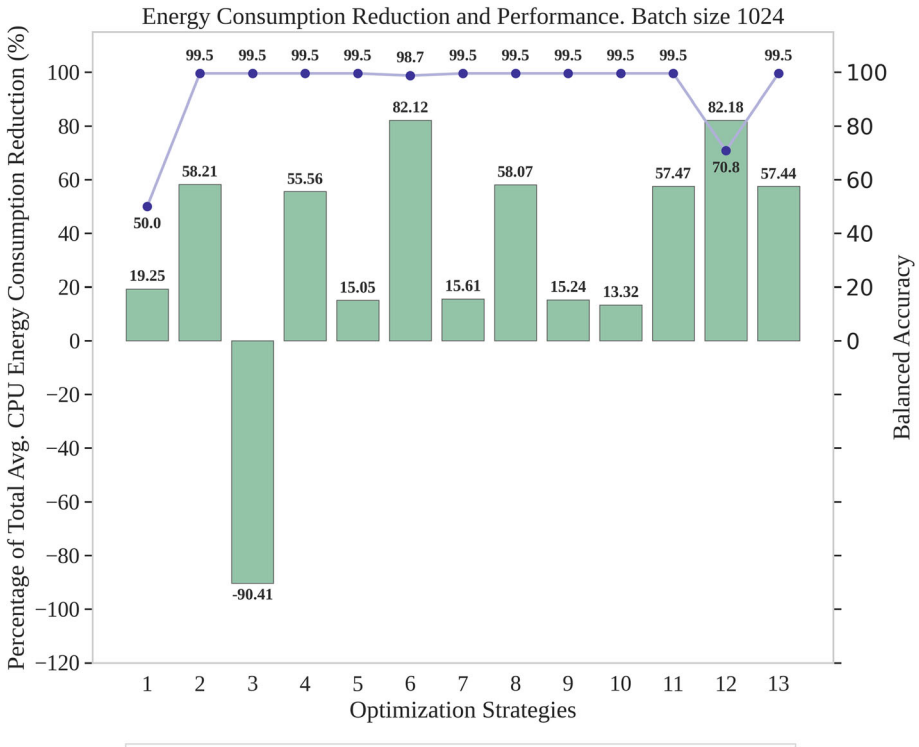

Fig. 5 Comparative analysis showing the energy efficiency and performance metrics of all optimization strategies evaluated with a large batch of 1024 samples

As in the medium batch size case, the results demonstrate that the optimization strategy selected by the energy efficiency and balanced profiles (the one consisting of the neural architecture search and the knowledge distillation technique) provides a model with significantly lower energy consumption than the baseline model (82.124%) while showing a very low-performance degradation with respect to the baseline model (0.08% for the balanced accuracy, 0.016% for the accuracy, and 0.11 for the F1-score). On the other hand, the optimization strategy selected by the performance optimization profile (the half-precision floating-point weight quantization technique) provides a model with the same performance as the baseline model and energy savings of 58.208% with respect to the energy consumption exhibited by the baseline model, more than double the 25% reduction in energy consumption that was established as the threshold of the optimization process.

The results obtained in terms of accuracy and balanced accuracy are also very favorable in general. With only two exceptions, the optimization strategies have managed to maintain a balanced accuracy above the threshold we set at the beginning of the experimental evaluation. In all cases, the accuracy and the F1-score were almost unaffected. However, due to the significant class imbalance in the data, this result is irrelevant, so we continue to focus on balanced accuracy. The largest balanced accuracy loss is obtained with the application of 8-bit integer weight quantization after training (0.5), followed by the balanced accuracy loss produced with the application of neural architecture search, knowledge distillation, and half-precision floating-point weight quantization (0.287) and, following after, the combination of neural architecture search and knowledge distillation (0.008).

6.4 Energy cost analysis of the model training and optimization

An interesting point to analyze is the energy consumption required to train the model that is the object of the optimization process and the time required to apply the optimization strategy to it. This consideration is especially important when designing systems that must be periodically retrained to incorporate new knowledge or learn from new data when the statistical properties of the data or the relationship between input and output variables change over time, since the time required for applying the selected optimization strategy will be incurred every time a retraining is needed. In the results shown in Table 6, it is observed that optimization strategies that have a higher energy cost when applied are those that involve a training procedure. That is, knowledge distillation, quantization-aware model fine-tuning, and pruning-aware model fine-tuning, in that order of increasing energy cost. This is because optimization strategies that involve a training procedure require more energy due to the computational intensity involved in the backpropagation process. Among these techniques, knowledge distillation is, by far, the most

Table 6 Total average energy consumption obtained by aggregating the measured values collected over multiple executions and over the duration of the experiment during the baseline model training and the optimization application

Opt. Strategy Id.	0	1	$2^{+\star}$	3	4	5	6	7	8	9	10	11	12	13
Total Avg. CPU Energy Consumption (J)	***275.193***	418.206	300.446	535.77	353.696	498.46	654.814 (95467.216*)	830.898	1035.579 (95847.981*)	1008.426 (95820.828*)	1362.88 (96175.282*)	372.319	766.375 (95578.777*)	873.113 (95685.515*)
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

$^{+}$ Optimization strategy that minimizes total average energy consumption

* Optimization strategy that minimizes total average energy consumption and satisfies the minimum acceptable threshold as measured by the user-defined performance target metric

* Considering the cost required for the neural architecture search procedure used to find the optimal model structure

Underlined values: Top three optimization strategy that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

J: Joules

expensive optimization strategy among these techniques, since is a computationally intensive process that explores a vast search space of potential architectures to identify the most efficient and effective neural network design for the specific task. This exhaustive search incurs substantial computational costs, contributing significantly to the overall energy consumption of the knowledge distillation technique. On the other hand, strategies that do not require a training procedure (such as post-training weight quantization) require less energy because no training is involved in the process, and the only task required is to process the model weights in a single pass to obtain the quantized version. In light of these results, it is possible to conclude that the use of optimization strategies involving a training procedure should be carefully considered when designing systems that need to be periodically retrained, such as those in the context of edge AI and those that are part of an online learning system. In that case, post-training weight quantization techniques may be the most efficient alternative.

7 Discussion

This section provides an integrated discussion focusing on key aspects of the framework. Section 7.1 presents a comprehensive overview of the study's primary findings and key takeaways. Section 7.2 delves deeply into the framework's scalability aspect. Lastly, Sect. 7.3 thoroughly examines the current limitations of the proposed framework while spotlighting potential directions for future research to overcome these limitations.

7.1 Summary of the results obtained

The main conclusions and observations derived from the results of the experimental evaluation we conducted in the framework within the context of the cryptomining use case are summarized below, organized by topic for ease of comprehension.

Best optimization strategies Given the existing trade-off between reducing energy consumption and avoiding degradation of model performance, different optimization strategies can be applied.

- The results show that, regardless of the batch size used for inference, an optimal balance between energy consumption and model performance can be obtained with the application of the neural architecture search and knowledge distillation techniques, as it provides very high energy savings with minimal degradation of performance. In our experiments, the use of neural architecture search followed by the knowledge

distillation technique provided the largest reduction in energy consumption in most cases studied, reducing the total average energy consumption by up to 82.304% with a minimal performance degradation of just 0.08% in the balanced accuracy, 0.016% in the accuracy and 0.11 in the F1-score.
- The results demonstrate that, when performance degradation is not allowed, the optimization strategy based on the application of the half-precision floating-point weight quantization provides the best energy-consumption results. In our experiments, the application of the half-precision floating-point weight quantization provided a reduction in energy consumption of up to 58.208%, with no performance degradation compared to the baseline model.

Combination of optimization techniques Some of the optimization techniques studied are not mutually exclusive and can be applied in conjunction with each other to further reduce the energy consumption of the model.

- Applying a weight quantization as the final post-processing can potentially reduce the energy consumption of the model in the inference stage. In particular, we observed that the optimization strategy based on the application of the neural architecture search and knowledge distillation techniques followed by a half-precision floating-point weight quantization provides a reduction in energy consumption of up to 80.741%, with a negligible performance degradation of 0.08% in the balanced accuracy, 0.016% in the accuracy and 0.11 in the F1-score. However, in some cases, we observed that the reduction in energy consumption was not appreciable, and therefore the cost-effectiveness of applying a post-training weight quantization technique should be evaluated on a case-by-case basis.
- Since pruning makes a significant portion of the model weights zero, if a post-training quantization technique is subsequently applied, it will only reduce the model size by a small factor and, in general, will do little to increase the model efficiency already achieved by applying the pruning technique.
- Pruning-aware model fine-tuning can be applied in conjunction with the neural architecture search and knowledge distillation techniques to further reduce the energy consumption of the model. The optimization strategy based on the application of the neural architecture search and knowledge distillation techniques followed by a pruning-aware model fine-tuning provides a reduction in energy consumption of up to 81.046% but with a significantly higher performance degradation of 0.287% in the balanced accuracy. However, in some preliminary tests, we observed that performance degradation was unpredictable, as this optimization strategy provided different performance results in different executions. For this reason, we recommend caution when applying this optimization strategy.
- Regarding quantization-aware model fine-tuning, when this technique is applied as a single optimization step, it provides a reduction in energy consumption of up to 15.05%, with no performance degradation. Although our results confirm the effectiveness of this technique in improving model energy efficiency when performance degradation is not permissible, the improvement in energy efficiency obtained is comparatively lower than the one obtained with other techniques, such as neural architecture search, knowledge distillation, or pruning-aware model fine-tuning. In addition, when this technique is applied as part of a two- or three-step optimization strategy in which neural architecture search, knowledge distillation, or the pruning-aware model fine-tuning technique are applied first, we observed that it did not result in a further reduction in energy consumption.
- Optimizing the quantization of neural network models is a challenging task due to the complexity of effectively handling mixed-precision operations. This challenge is especially evident in cases where weights are quantized to 8-bit integers, while activations, which produce the outputs of hidden layers, are kept to 16 bits to avoid significant loss of precision in internal representations. However, the lack of optimization for this mixed-precision approach, where operations on different data types must be optimized together, results in a significant increase in energy consumption and slower inference in models produced with this quantization scheme. In fact, these models can be up to 1.84 times slower than the original model or other quantization schemes such as integer weight quantization and half-precision floating-point quantization, resulting in inference that is, respectively, 2.1 times and 3.72 times slower. Nevertheless, integer weight quantization with 16-bit integer activations and 8-bit integer weights can still be advantageous as it reduces the model size and results in faster loading times. Furthermore, as observed in the results presented above, the higher precision of the activations prevents a significant loss of accuracy in the internal representations compared to 8-bit integer quantization, which is crucial for some applications. In the future, support for kernels specifically optimized for this type of quantization scheme in TFLite could potentially resolve the current performance issues observed.

Batch sizes

- In most cases, optimization strategies that provide the greatest reduction in energy consumption for small batch sizes also provide the greatest reduction for large batch sizes, and when this is not the case, the difference in energy savings is small. However, it is clear from the results that the reduction in energy consumption obtained with the application of optimization strategies is greater for larger batch sizes, regardless of the optimization strategy applied. From this observation, we can conclude that while it is true that larger batch sizes provide higher energy efficiency compared to the non-optimized model using the same batch size, there may be cases where smaller batch sizes actually consume less energy. For example, if the non-optimized model using a large batch size requires substantially more energy than the non-optimized model using a small batch size, then the reduction obtained with an optimization strategy on the large batch size may lead to net energy consumption that is actually greater than the net energy consumption obtained with the same optimization strategy on the small batch size. That is, although larger batch sizes typically provide greater energy efficiency, there may be cases where smaller batch sizes are preferable for optimal energy savings, and therefore the choice of batch size should be carefully considered to ensure that net energy consumption is minimized.

Model size

- Knowledge distillation facilitates a notable reduction in the model size while nearly completely mitigating performance degradation. This method enables considerably greater model compression compared to the naive approach of simply reducing parameters. In our case, it even allowed further reduction of the model size, although the performance of the optimized model becomes very inconsistent across different trials, failing to correctly predict the minority class on some occasions. However, this might not be a problem for model-specific optimization, which will certainly compensate for the cost of the several trials needed to obtain a well-performing model in the medium term. On the other hand, we have found that when training a model of the same size as the one used to perform the knowledge distillation technique, the model performs poorly. This allows us to conclude that knowledge distillation allows the use of very small models that would otherwise not be usable due to lower-than-acceptable classification accuracy and therefore could not be implemented in production. When the temperature factor is increased until the optimum point is reached progressively, the performance of this model is observed to increase, nearly approximating the performance obtained with the baseline (teacher) model.

Model optimization process

- An important detail is that when using an imbalanced dataset, it is necessary to pay special attention to the choice of hyperparameters used to apply the optimization techniques. In our use case, we have found that pruning-aware model fine-tuning and quantization techniques are particularly sensitive to the choice of these values when dealing with imbalanced datasets. In that case, if the hyperparameter values are not chosen carefully, performance degradation after optimization can be severe. Conversely, when the classes in the dataset are balanced, we have found that these techniques are more robust against the choice of these hyperparameters than in the case of using an imbalanced dataset, since the performance is generally maintained above the minimum acceptable through different retrainings. Moreover, the case of the strategy that consists of applying neural architecture search and knowledge distillation is similar. The model can be reduced by a larger factor if the number of samples in each class is balanced, but if it is not, the model can be reduced less with respect to the original model. In the latter case, if the model is reduced excessively, the accuracy in predicting the minority class will drop to a random estimate or, in the worst case, to zero (i.e., the model overfits to predicting the majority class), although, in some cases, it is still possible to find a local minimum in some training attempt to obtain acceptable accuracy for both classes. However, the likelihood of this occurring decreases as the model is reduced further.

Temporal Analysis of the Framework Execution

- To evaluate the efficiency and usability of our proposed framework and provide a better understanding of its computational demands, the execution times for all optimization strategies during training, inference, and load phases have been carefully measured. This comprehensive analysis of the temporal aspect of running the framework is presented in Table 9 within the Appendix. This table delineates time statistics across five executions for each optimization strategy. It includes key metrics such as average, standard deviation, and maximum times for training, inference, and load processes. Section 9.2.2 thoroughly discusses these results, extracting meaningful conclusions to underscore their significance in our framework's evaluation. It should be noted that the total execution time will vary depending on the machine specifications where the

framework is running. However, in our experiments, which were executed on a standard desktop computer with specifications listed in Sect. 6.1, the total execution time for the complete framework, including all optimization profiles and strategies and three different batch sizes amounted to a total of three hours and twenty minutes approximately in our case (approximately one hour per batch size evaluated and an additional twenty minutes for the initialization of the framework). Given the moderate capabilities of the machine used in these experiments, the overall time taken for the optimization process remained relatively short. For significantly more complex models or models that require quicker updates, a more powerful machine to run the framework might be necessary, or, alternatively, employing horizontal scalability might be another solution if vertical scalability is not possible. However, it's worth noting these models that require such frequent updates are uncommon in practice, especially when complex models are involved. Moreover, in practice, highly complex models usually don't face significant resource constraints, so energy consumption in those cases is usually not a critical factor. Hence, we believe that the time spent is generally a worthwhile investment, considering the significant advantages in terms of energy efficiency and achieving a higher rate of inference. Future enhancements to the implementation of the framework or its supporting libraries could further increase its effectiveness in handling these scenarios more efficiently.

7.2 Considerations on the scalability of the proposed framework

Regarding the scalability of the proposed framework, especially in the context of large-scale data center scenarios involving concurrent operation of multiple DNN models, a thorough explanation of this matter is essential. Therefore, in this section, we will delve into the key aspects of scalability within our proposed framework for continuous optimization of multiple DNN models in large-scale applications.

7.2.1 Application of the framework for simultaneous model optimization

Our framework operates on a per-model basis, which implies that for simultaneous optimization and execution of different DNN models, the framework must be run independently for each model. In contrast, when only a single model is used, the optimization process takes place only once for each time the model needs to be optimized, and, once optimized, in the case of distributed applications, the model can be deployed to the desired locations or optimized in-place. Alternatively, if possible, a centralized batch inference method can also be used, in which the data is aggregated in a central location for batch inference using a single model. In this regard, a comprehensive discussion of batch inference is detailed in Sect. 6.3, accompanied by supporting data in Tables 3, 4 and 5.

7.2.2 Simultaneous execution of multiple models

For concurrent execution of the framework on multiple models, certain metrics collected by our framework operate at the system level. Therefore, ensuring the reliability of these metrics requires the exclusion of concurrent processes in the same physical machine. To account for this, we have designed our framework to verify this requirement prior to the start of the optimization by requiring its fulfillment before starting the optimization process. Consequently, in its current form, parallel optimization of numerous models using our framework can only be achieved through horizontal scalability by running the framework on multiple physical machines simultaneously. It should be noted that virtualization is not a valid approach as Intel RAPL does not provide container-level power consumption measurements when running within a virtualized environment, and, instead, global power consumption is reported. This limitation poses a challenge to our framework's scalability within virtualized infrastructures, as it prevents precise measurement of the individual energy consumption of each optimized model, which completely prevents accurately and reliably selecting the best optimization strategy for each model that needs to be optimized. Additionally, this limitation also poses a challenge for deploying our framework in cloud-based or virtualized infrastructures where direct access to physical hardware components is not possible or it is shared among multiple tenants.

7.3 Limitations and challenges of the proposed framework

This section critically evaluates the framework's limitations and the obstacles that might hinder its adoption and effectiveness in certain real-world scenarios. First, the impact of the model inference rate on overall energy efficiency is analyzed. Next, the overall energy cost and efficiency for different types of application architectures are evaluated. Finally, the limitations regarding the framework's applicability for the continuous optimization of models deployed in dynamic environments are discussed.

7.3.1 Impact of model inference rates on energy efficiency

The proposed framework faces some limitations that require careful consideration, especially in scenarios characterized by variable inference rates and dynamic conditions to which the model must adapt. More specifically, continuous optimization of one or several models in environments with frequently changing conditions, such as in distributed edge inference scenarios or settings where federated learning is employed, can be challenging.

This is especially the case when the inference occurs infrequently, and the energy cost savings obtained from optimization may not offset the total energy invested in executing the optimization process for a specific model. Therefore, the frequency of inference, determined by the specific usage scenario, and the energy invested to optimize the model to be deployed are key factors to evaluate if minimizing total energy consumption is considered important for the application. To illustrate this point, we draw upon some of our previous research and explore two typical ML/DL inference scenarios to shed light on the challenges and complexities associated with applying our optimization framework in scenarios of considerably different inference rates per second in production environments.

- *Low inference rate scenario* In prior research work [49], the focus was placed on developing and implementing chlorophyll-a soft-sensors adjusted for situations with limited computational capacity and constrained resources utilizing automatic high-frequency monitoring systems and ML techniques to create a cost-effective and rapid tool for estimating chlorophyll-a fluorescence, especially beneficial for supporting manual sampling in water bodies susceptible to harmful algal blooms. This involved the development of compact ML models designed for low-energy consumption and efficient performance, specifically targeting deployment on buoys with restricted hardware and battery resources. The inference, in this case, occurred at a notably low frequency, spanning from once a day to a prediction horizon of seven days into the future. The low frequency of these inference operations will pose a challenge regarding the energy consumption optimization of these models. The computational expense associated with optimizing these models for better energy efficiency using our proposed framework is expected to demand significant energy consumption. However, amortizing this energy cost against the infrequent inference process once the model is deployed will potentially present a scenario where the gain achieved in energy cost will take considerable time to offset the total energy investment required in the optimization process. Furthermore, if we factor in relatively frequent model updates to address potential issues like data drift or changes in the environment, the increased frequency of the optimization process could worsen the gap between the energy spent on optimization and the infrequent cases where inference operations are used. As a result, these frequent updates may not effectively balance out the energy costs linked with optimization, possibly leading to minimal or negative overall energy savings.
- *High inference rate scenario* The case study presented in Sect. 5 focuses on a high-rate inference scenario exemplified by a real-world use case related to cryptocurrency mining detection that requires highly efficient and accurate DNN models for real-time classification of network traffic. These models are deployed at several remote locations, each location serving as an aggregator for a subset of the global network data. Therefore, a considerably high rate of inference processing is expected for each deployed model. The security-critical nature of this application underscores the need for high-throughput, low-latency models that efficiently identify cryptocurrency mining activity within a substantial volume of network data, while operating under significant resource constraints, including the unavailability of specialized AI accelerators, low computational resources, and minimal energy consumption. In this case, due to the high inference rate that is expected, energy efficiency is critical for efficient, sustainable, and cost-effective operation. Therefore, the high frequency of inference operations in this scenario allows for the energy savings obtained from the optimized models to offset the energy expended during the optimization process relatively quickly. Continuous adaptation and updates to the model might still be necessary due to the evolving nature of cryptocurrency mining methods, network behaviors, and potential adversarial attacks. However, if the expected inference rate in the production environment is sufficiently high, as in the use case selected for the present study, the continuous energy savings achieved through energy-efficient inference will justify the investment in the recurring optimization cycles triggered by the necessary model updates.

The significant energy savings achieved during the inference processes—once the model is deployed in production—will eventually offset the initial time and energy investment, assuming that the model is not updated too frequently to negate these gains. The inference rate of the model will ultimately determine the rate at which this break-even point is reached. Therefore, evaluating the applicability of the proposed framework in various

inference scenarios when minimizing total energy consumption is a significant concern and requires a detailed understanding of the trade-offs between model optimization and expected inference frequencies in the production environment.

However, although the initial energy cost associated with the optimization process provided by our framework is significant, especially for the more complex optimization techniques, as noted in Sect. 6.4, and should be taken into account when minimizing the total energy cost is important for the specific application, the energy efficiency gains that can be achieved in the inference process by using our framework take precedence over the overall energy cost within edge inference scenarios where IoT devices operate under energy constraints imposed by the use of limited capacity batteries.

The resulting energy savings during the inference process are crucial for extending the battery life of these devices, which offers substantial advantages in terms of maintenance cost savings, which is especially relevant in cases where the IoT devices are situated in remote locations such as the scenario described above where several buoys are deployed in distant bodies of water. Additionally, it enables the use of lighter, lower-capacity batteries that can help reduce deployment and maintenance costs.

7.3.2 Evaluation of the energy costs and efficiency in centralized and edge inference scenarios

As mentioned above, the optimization process within our framework is exhaustive and incurs a certain energy cost, especially for techniques involving training processes, as evidenced in Table 6 and elaborated in Sect. 6.4. For this reason, in scenarios where the edge computing paradigm is not employed, the feasibility of deploying our framework for continuous optimization in purely isolated IoT environments where embedded systems are used with a limited computer might be limited.

Nevertheless, the application of our proposed framework provides significant benefits in scenarios where centralized inference, such as the case study presented in Sect. 5, or edge computing is considered in the application's architecture is feasible. To illustrate the latter, in prior research [50–53], we focused on utilizing multi-access edge computing (MEC) for remote trajectory control of automated guided vehicles (AGVs). The MEC server facilitated periodic training of multiple ML/DL models through transfer learning. These trained models were then deployed to simultaneously control the trajectory of several vehicles. Our framework allows continuous optimization of these models within the MEC platform, benefiting from ample computational power without facing energy consumption constraints. As a result, the application of our framework in this context would result in substantial energy savings, especially considering the frequent inference requirements of ten times per second for each active vehicle in the factory, and even more so in medium to large factory environments with tens or even hundreds of AGVs often operating simultaneously.

7.3.3 Optimizing energy efficiency in dynamic environments

Our proposed framework addresses the challenge of managing dynamic changes in workloads or data by re-running the optimization process whenever updates to the model are required. This iterative approach ensures that the model remains optimized while being continuously aligned with the evolving data distribution and the specific workloads it encounters in its deployment environment, which is crucial in sustaining high-performance levels over time.

Regarding the frequency of optimization cycles, as we mentioned earlier, excessively frequent changes can lead to substantial computational overhead. This concern becomes particularly pertinent in real-world scenarios characterized by frequent dynamic changes, such as in distributed edge inference or federated learning environments.

Moreover, as also mentioned above, in scenarios where inference processes occur infrequently, the total energy invested in frequent model updates may require considerable time to compensate. This is especially true when considering the energy expenditure associated with these frequent updates.

However, in the context of distributed edge inference, as mentioned above, the emphasis is on prioritizing the reduction of energy consumption during inference processes rather than minimizing total energy costs, especially when inference is performed using IoT devices operating under strict battery constraints. Therefore, in this context, despite considering the need for these recurring optimization cycles, the improved energy efficiency in the inference process achieved by our framework provides important benefits regarding extended battery life and reduced deployment and maintenance costs.

8 Conclusions and future work

In this article, we propose an integrated methodological framework to analyze the energy consumption and resource utilization of DNN-based systems deployed in production environments. The proposed framework allows us to evaluate and compare the performance of the combination of a variety of state-of-the-art model optimization techniques and provide a deep understanding of the trade-offs between energy efficiency, resource utilization, and

performance. In addition, the framework proposes the concept of an "optimization profile" to address the trade-off between energy savings and model performance. Optimization profiles allow to reflect in a quantitative way scenarios in which energy savings or model performance should be balanced or may take priority over the other.

Our proposed framework for energy-efficient optimization of deep neural networks (DNNs) provides added value to the optimization process by automating the process of finding optimal hyperparameters and configurations of the network architecture that minimize energy consumption without compromising model accuracy. This streamlines the optimization process, saving valuable time and resources and allowing researchers and practitioners to focus on other aspects of their projects, such as data preprocessing and model interpretation.

The application of our automatic framework to optimize the energy efficiency of DNNs can improve the reproducibility of the results by ensuring consistency between different experiments, allowing fair and robust comparisons.

It is worth noting that typical approaches based on trial-and-error optimization of DNN energy efficiency can be sub-optimal and inefficient, which underscores the importance of the automated framework we propose in this study. Our framework defines a comprehensive set of predefined common "optimization strategies" that combine multiple state-of-the-art optimization techniques that can be applied together, enabling fast and efficient optimization of a given model. The integration of an extensive selection of optimization techniques that can be automatically applied and evaluated to identify the strategy that leads to maximum energy efficiency and minimizes resource consumption for a specific model facilitates adaptability across diverse datasets and model architectures. By establishing a systematic process that exhaustively evaluates the available optimization strategies based on real-time energy consumption and performance feedback at inference time, our proposed framework alleviates the burden associated with trial-and-error optimization methods and effectively mitigates the potential pitfalls frequently observed in the application of these manual optimization processes, while ensuring a more robust and efficient optimization process, that, due to its automated nature, can be seamlessly integrated into continuous integration pipelines to fully harness the advantages offered by the state-of-the-art energy optimization techniques available across diverse deployment scenarios. The variety of model optimization techniques encompassed in this broad set of optimization strategies underscores the versatility of our proposed framework to adapt to various model optimization paradigms.

Moreover, the flexibility offered by the framework's modular design facilitates the integration of new optimization techniques and customized strategies into the existing framework architecture to address more specific requirements. The adaptability and extensibility of our framework are fundamental features that enable researchers to readily incorporate novel optimization methods or those that already exist and were not initially included in our work, thereby perpetually expanding the repertoire of optimization methodologies available within the framework. In this way, users of the framework can benefit from the latest advances in optimization techniques without being constrained by the framework's pre-existing functionality, thus perpetuating its relevance and usefulness in the ever-evolving landscape of model optimization research.

Importantly, this framework is open source and publicly available (github.com/amitkbatra/EnergyNet), providing an excellent opportunity for contributions from the scientific community. These contributions could incorporate new techniques or enhance those already included, thus establishing the framework as the foundational basis for energy efficiency optimization in DNNs. By providing access to a variety of optimization tools and techniques, this framework offers an opportunity for the scientific community to contribute to the improvement of energy efficiency in DNNs, making it a collaborative effort toward sustainable artificial intelligence.

The effectiveness of our proposed framework was demonstrated by applying it to a real-world use case, namely cryptomining detection, and successfully optimizing the energy efficiency of a DNN, achieving an 82.304% reduction in total energy consumption while maintaining a minimal performance loss of 0.008%. Our results demonstrate that our framework is a reliable and effective way to analyze and optimize the energy efficiency and resource utilization of DNNs. Through careful analysis of the trade-offs between energy efficiency and performance, our framework allows us to automatically select the most appropriate optimization techniques according to the specific application requirements.

As future work, we plan to extend our proposed framework to enable the analysis of the energy efficiency and resource utilization of different DNN architectures, such as recurrent neural networks and convolutional neural networks, as well as different ML training methods (e.g., reinforcement learning). In addition, we plan to extend our framework to analyze the energy efficiency of DNN-based systems on specialized hardware platforms, such as GPUs and TPUs. To that end, other inference engines that provide support for the required platforms, such as TensorRT for the GPU case, should be added. On the other hand, providing support for evaluation in different hardware architectures, such as heterogeneous and multi-core systems, as

well as in distributed systems, such as clusters and cloud computing, is also an interesting line of future work.

Moreover, to address computational overhead, various strategies to optimize the efficiency of subsequent model optimization processes within our framework are being actively explored. One of our lines of research focuses on the development of more efficient methods for incremental model updating. Leveraging the latest advances in transfer learning techniques, the goal is to enable the framework to update and adjust models more efficiently. By leveraging pre-existing model knowledge and parameters, this approach significantly reduces the computational burden associated with large-scale re-optimization.

In addition, we plan to incorporate automatic data drift detection mechanisms into the framework. Continuous monitoring of incoming data characteristics allows us to identify shifts in data distribution that might affect the model performance. This proactive approach assists in determining precise times for model re-optimization, ensuring timely adjustments. Quantifying data drift involves analyzing statistical measures and metrics to measure the degree of deviation between the most recent data and the original training data. These metrics encompass statistical distances, divergence measures, or similarity scores that facilitate the comparison of data distribution changes. To streamline this process, we empirically evaluate a reference point for expected data similarity, allowing for the definition of adjustable thresholds. These thresholds trigger automatic optimization when exceeded, ensuring that the model remains aligned with evolving data patterns. In addition, quantifying this deviation also helps determine the degree of model retraining needed, which can be used to further optimize the re-training efficiency.

Finally, we plan to explore alternative selection methods for improving the combinatorial search space for optimization strategies. In contrast to the exhaustive search method currently employed, we propose to employ a greedy strategy. This approach consists of systematically applying optimization techniques to the target model one at a time, selecting and retaining the technique that yields the most favorable results aligned with the chosen optimization profile. This iterative process continues until no further improvements can be achieved. Although this approach represents a suboptimal solution, as it assumes that the best combination of techniques emerges through consecutive application of optimization techniques that lead to incremental improvements, preliminary tests have shown promising results. Other strategies, such as Monte Carlo tree search and genetic algorithms, are also being explored for their potential to effectively navigate the vast search space for optimization strategies, which will become increasingly larger as the repertoire of optimization techniques available in the framework expands over time.

Appendix

Analysis of resource usage in inference time

The use of computing resources (e.g., CPU, memory, and disk) has a direct impact on energy consumption. In this section, we analyze in detail the best optimization strategies that reduce the use of computing resources, globally considering all optimization profiles and taking into account the prior selection of an optimization profile (energy efficiency, balanced, and performance profiles). The analysis is performed considering different batch sizes during the inference process.

Analysis of CPU consumption

From a general perspective, analyzing the results obtained in the inference phase using the three batch sizes, some interesting conclusions can be drawn about the resource usage and the performance obtained by the models in the evaluation phase. An interesting result is that no reduction in CPU power consumption is achieved by applying any of the optimization strategies when using a batch size of 32. In fact, the results indicate that, in some cases, optimization strategies lead to an increase in CPU power consumption. The reduction in power consumption of the optimized models is mainly due to the reduction in total inference time as the number of operations and their required computation time decrease. This is because an energy optimization strategy consisting of reducing the number of operations to be executed or reducing the number of parameters to be used to perform the inference step of the ML process results in a reduction in the time required to perform the inference step. This reduction in total inference time translates directly into a reduction in total energy consumption needed to perform the inference step, since this energy consumption is directly proportional to the time needed to perform the inference step. The reason why no reduction in CPU power consumption is achieved when using a small batch size seems to be related to the fact that the total inference time is so short (even shorter after optimization) that the CPU power consumption is not visibly affected. However, we can see that a small reduction in CPU power consumption is achieved when the batch size is increased. Overall, the null reduction in CPU power consumption observed in Table 3 and the small decrease observed in Tables 4 and 5 is not a concern, since the reduction of the energy consumption of the target model during the inference state is the main goal, as this is the key factor to minimize when optimizing the energy efficiency of the target model and, as can be seen from the results presented in Table 3, most optimization strategies

provide a high reduction in energy consumption with respect to the non-optimized model.

Analysis of CPU frequency

Another interesting conclusion that can be drawn from the results presented in Tables 3, 4 and 5 is that some optimization strategies lead to a consistent reduction in CPU frequency during the inference stage of the optimized model, regardless of the batch size used to perform the prediction. This is the case for the optimization strategy consisting of applying pruning-aware model fine-tuning and the one consisting of applying neural architecture search and knowledge distillation. In both cases, the reduction in frequency is due to the fact that the application of these techniques leads to a reduction in the number of operations to be performed by the CPU to carry out the inference. In the case of model pruning, the reduction in the number of operations is due to the use of sparse matrices in the calculation of the operations performed by the model, which can be accelerated by using appropriate optimization algorithms. In the case of the strategy of neural architecture search and knowledge distillation, the reduction in the number of operations is due to the smaller number of parameters to be evaluated in the inference stage, which translates directly into a reduction in the number of operations to be performed.

Finally, we can see that CPU utilization was hardly affected by any of the optimization strategies, so the results presented in Tables 3, 4 and 5 show that the percentage of CPU utilization remains virtually constant regardless of which optimization strategy was applied.

Analysis of memory usage

Regarding memory usage, we observe in the results presented in Tables 3, 4, and 5 that all optimization strategies that have been applied in the inference stage lead to a consistent but small reduction in memory usage, regardless of the batch size used to perform the prediction. A reduction in memory usage can be potentially obtained by reducing the number of parameters (in the case of applying knowledge distillation or pruning techniques) or the size of the parameters in memory (in the case of optimization strategies involving weight quantization techniques). Although not measured, this reduction in memory consumption also contributes to reducing the amount of energy required to carry out the inference stage of the ML process, since the smaller the number of data to be stored in memory, the lower the amount of energy required to perform data storage and retrieval operations. In addition, this reduction in memory usage also increases the inference speed of the optimized model, since the number of read operations to load parameters from the main memory to the cache and CPU registers is reduced. Finally, minimizing memory usage is crucial to allow more models to be deployed on the same device, reducing infrastructure costs, and improving system scalability. In this case, the most favorable results in terms of reduced memory usage are obtained with the application of 8-bit integer weight quantization, leading to a reduction of approximately 0.33% in memory usage relative to the baseline model. However, if we ignore this technique due to the considerable reduction in balanced accuracy it causes, the optimization strategy that provides the most favorable results in terms of memory usage reduction is the optimization strategy consisting of applying half-precision floating-point activations and quantization of 8-bit weights, the one consisting of applying model fine-tuning with quantization, and the one consisting of applying neural architecture search and knowledge distillation, three of which provide between 1.5% and 3% memory usage reduction. As can be seen, the reduction in memory usage obtained with the application of weight quantization techniques is relatively low, with the exception of full quantization of 8-bit integer weights. This is due to the fact that half-precision floating-point weights are not natively supported by standard desktop CPUs, which means that the optimized model parameters will have to be converted to a single-precision floating-point format to be processed by the hardware. This conversion to a single-precision floating-point format will ultimately increase the total size of the parameters stored in memory. The small reduction observed is due to the elimination of some redundancy during the lossy compression that occurs when the parameters are converted from the half-precision floating-point format to the single-precision floating-point format. However, if instead of using the CPU, a specialized accelerator with hardware support for half-precision floating-point parameters is used to perform the inference, such as an NVIDIA GPU, Google's TPU or NVIDIA Jetson embedded computing boards, it will be possible to operate directly on the half-precision floating-point parameters without incurring the overhead of converting them to single-precision floating-point format. This will result in significantly lower memory usage, leading to lower power consumption during the model inference stage, and will improve the scalability of the system, as potentially more models can be deployed on the same device. On the other hand, this problem is not observed with the optimization strategy based on the application of full 8-bit integer weight quantization, since integer matrix multiplications are natively supported by most hardware architectures, including standard desktop and mobile CPUs.

Table 7 Size of the compressed (Deflate) model file in persistent storage obtained with each optimization strategy

Opt. Strategy Id.	0	1	2	3	4	5	$6^{+\star}$	7	8	9	10	11	12	13
Model Size (B)	13,194	12,415	45,977	12,722	27,929	12,028	***1660***	10992	27,917	12,326	11,345	27,909	1662	27,886
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

$^{+}$ Optimization strategy that minimizes model size

* Optimization strategy that minimizes model size and satisfies the minimum acceptable threshold as measured by the user-defined performance target metric

Underlined values: Top three optimization strategy that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

B: bytes

Analysis of disk usage

Also, the size of the model on disk is another factor that influences the energy consumed by the model. When a model is deployed in a production environment, it must be stored on a persistent storage medium, such as a flash drive or hard disk. The smaller the size of the model, the less energy is required to store it and the less energy is required to load it from persistent storage to main memory. Optimizing the size of the model in terms of the number of parameters also allows maximizing the number of models that can be stored on a given persistent storage medium, thus eliminating the need to add new storage devices each time new models are produced, which also results in lower energy consumption. In the results presented in Table 7, the size of the compressed model file in persistent storage is included. In this table, we can observe that the size of the non-optimized model on disk is 13,194 bytes, while the smallest optimized model obtained after applying the optimization strategy consisting of applying neural architecture search and knowledge distillation to the model has a size of only 1660 bytes, which is an 87.42% reduction in model size. The second smallest optimized model is the one obtained by applying the neural architecture search and knowledge distillation optimization strategy to the model followed by a half-precision weight quantization, which is a slightly bigger model with a size of 1808 bytes. The third smallest optimized model is the one obtained by applying a pruning-aware fine-tuning technique to the model followed by a quantization-aware model fine-tuning technique, which has a size of 10,992 bytes, which is 16.68% smaller than the baseline model.

Analysis of additional results

Analysis of the results obtained in the load stage

The results obtained in the load phase are shown in Table 8. Interestingly, energy efficiency in this phase is slightly degraded by all optimization strategies. Only two strategies notably improve energy efficiency, which are pruning-aware model fine-tuning followed by quantization-aware model fine-tuning and the one consisting of neural architecture search and knowledge distillation followed by half-precision floating-point weight quantization. Both strategies improve the energy efficiency of the model load by approximately 3%. The combination of neural architecture search, knowledge distillation, and pruning-aware model fine-tuning also improves energy efficiency, but to a lesser extent (0.376%). These results suggest that improving the energy efficiency of the model loading process using these techniques provides small benefits and that for specific scenarios where models are loaded frequently, such as mobile applications, these gains may be difficult to justify and alternative techniques, such as using more efficient hardware designs, may be more beneficial.

Analysis of the inference, training, and load times

The results obtained in the training, loading, and inference stages of the optimized models are summarized in Table 9. The results shown in this table correspond to the mean, standard deviation, and maximum time (in seconds) obtained at each stage over five iterations for each optimization strategy. The most important result that can be observed in this table is the reduction of the inference time of the optimized models with respect to the baseline model. Achieving a high speed-up in the inference time is essential

Table 8 Total average energy consumption obtained by aggregating the measured values collected over multiple executions and over the duration of the experiment during the model load

Opt. Strategy Id.	0	1	2	3	4	5	6	7+⋆	8	9	10	11	12	13
Total Avg. CPU Energy Consumption (J)	***0.383***	0.389	0.393	0.393	0.402	0.391	0.395	**0.369**	0.382	0.4	0.384	0.395	0.372	0.391
Percentage of Total Avg. CPU Energy Consumption Reduction (%)	*N/A*	−1.459	***−2.541***	***−2.546***	***−4.759***	***−1.853***	−2.946	***3.668***	***0.376***	***−4.271***	***−0.136***	***−2.914***	2.97	***−1.933***
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

+ Optimization strategy that minimizes total average energy consumption

⋆ Optimization strategy that minimizes total average energy consumption and satisfies the minimum acceptable threshold as measured by the user-defined performance target metric

Underlined values: Top three optimization strategy that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

J: Joules

from a scalability point of view when the system must process large amounts of data in real-time. In this context, increasing the number of predictions the system can perform per second can have a significant impact on system performance, reducing the need to add more resources to the system to meet the real-time deadline required by the application, and thus reducing infrastructure costs and power consumption. The results presented in Table 9 show that the optimization strategy that minimizes model inference time is the one based on the neural architecture search and knowledge distillation techniques followed by pruning-based model fine-tuning, which speeds up model inference by 3.78 times. However, this optimization strategy does not meet the minimum performance that was set as a threshold for the optimization process (0.9 for balanced accuracy) and is therefore discarded in favor of the optimization strategy based on half-precision floating-point weight quantization, which accelerates model inference by 2.02 times with respect to the baseline model and without any performance degradation.

Another interesting point to analyze is the time required to train the model that is the target of the optimization process and the time required to apply the optimization strategy to it. In the results shown in Table 9, it is observed that the optimization strategies that have a higher training cost are the ones that include the knowledge distillation technique, as this strategy requires a neural architecture search procedure to find the most suitable structure for the student model. Although the time required to complete this search is a one-time process to be executed offline, it should be taken into account as it can potentially involve a large number of training processes and decrease or even nullify any potential energy efficiency gains that may be achieved by the optimization strategies during model application. However, this problem can be solved by manually finding a suitable student model architecture or at least alleviated by reducing the search space through manual experimental analyses. On the other hand, pruning-aware and quantization-aware model fine-tuning techniques are much faster than knowledge distillation, since the fine-tuning process to be performed by these two techniques is much faster than training the model from scratch and they do not require a neural architecture search procedure as the knowledge distillation technique, thus the total time required to apply these techniques is considerably less. Finally, post-training weight quantization techniques are also much faster than knowledge distillation and similar to fine-tuning techniques, being also unaffected by the high variance caused by the neural architecture search and training procedures.

Additionally, the reduction of loading times is also of great importance when the application requires a dynamic instantiation of the models or when the system must be able

Table 9 Average, standard deviation and maximum of training, inference, and load times averaged obtained for each optimization strategy across five different executions

Opt. Strategy Id.	0	1	2	3	4	5	6★	7	8	9	10	11	12+	13
Avg. Inference Time (s)	0.318	0.273	0.157	0.584	0.159	0.28	0.088	0.278	0.159	0.286	0.287	0.161	***0.084***	0.164
Std. Dev. Inference Time (s)	0.007	0.005	***0.002***	0.012	0.007	0.006	0.004	0.008	0.01	0.018	0.017	0.006	0.006	0.009
Max. Inference Time (s)	0.329	0.281	0.158	0.603	0.166	0.292	0.092	0.288	0.177	0.322	0.321	0.171	***0.09***	0.179
Avg. Training Time (s)	10.553	16.034	11.533	20.407	13.773	19.259	25.125 (4217.254*)	31.593	39.864 (4231.993*)	38.071 (4230.2*)	52.038 (4244.167*)	13.812	28.97 (4221.099*)	33.282 (4225.411*)
Std. Dev. Training Time (s)	0.156	0.045	0.03	0.071	0.085	0.089	6.452 (58.734*)	0.153	4.661 (58.607*)	2.609 (58.646*)	3.376 (58.548*)	0.032	21.148 (62.251*)	4.634 (58.732*)
Max. Training Time (s)	10.864	16.119	11.58	20.484	13.922	19.357	35.602 (4271.508*)	31.767	45.812 (4281.718*)	41.614 (4277.52*)	57.85 (4293.756*)	13.84	71.073 (4306.979*)	39.397 (4275.303*)
Avg. Load Time (s)	0.086	0.062	0.065	0.063	0.065	0.071	0.064	0.069	0.061	0.064	***0.06***	0.062	0.062	0.063
Std. Dev. Load Time (s)	0.005	***0.002***	***0.002***	***0.002***	0.003	0.015	***0.002***	0.015	0.006	0.004	0.003	0.003	0.006	0.003
Max. Load Time (s)	0.092	0.066	0.068	0.066	0.071	0.101	0.067	0.098	0.072	0.069	***0.065***	0.067	0.073	0.068
Accuracy	1	0.996	1	1	1	1	0.984	1	1	1	1	1	0.997	1
Balanced Accuracy	0.995	0.5	0.995	0.995	0.995	0.995	0.987	0.995	0.995	0.995	0.995	0.995	0.708	0.995
F1-score	1	0.993	1	1	1	1	0.989	1	1	1	1	1	0.997	1

+ Optimization strategy that minimizes average inference time

★ Optimization strategy that minimizes average inference time and satisfies the minimum acceptable threshold as measured by the user-defined performance target metric

∗ Considering the time required for the neural architecture search procedure used to find the optimal model structure

Underlined values: Top three optimization strategy that resulted in the best improvement in the measured statistic

Bold and underlined values: Optimization strategy that resulted in the best improvement in the measured statistic

Double underlined values: Optimization strategy that results in a model with performance above the minimum acceptable performance threshold

s: seconds

to change the model it uses in real-time, especially when the number of models it must handle simultaneously is large. In this sense, in Table 9, we can observe that all the optimization strategies that have been applied lead to models that can be loaded faster than the baseline model. The optimization strategy that allows the optimized model to load faster is the one based on the application of neural architecture search and knowledge distillation, followed by pruning-aware model fine-tuning and quantization-aware model fine-tuning as the last step. This optimization strategy speeds up the model loading time by 1.43 times relative to the baseline model.

Acknowledgements This work was partially supported by the European Union's Horizon 2020 Research and Innovation Programme under Grant 101015857 (TeraFlow), and through the HORIZON-JU-SNS-2022 ACROSS project with Grant Agreement number 101097122, and the Spanish Ministerio de Asuntos Económicos y Transformación Digital, Programa UNICO under project B5GEMINI-AIUC (TSI-063000-2021-79).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data availability Access to the repository containing the experimental data generated in this study is available at github.com/amitkbatra/EnergyNet. The dataset used in this study originates from Telefónica I+D, but certain restrictions limit their public availability. This dataset was utilized under a specific license for this study and is therefore not accessible to the general public. Nevertheless, they can be obtained from the authors upon a reasonable request, provided that permission from Telefónica I+D is granted.

Code availability Access to the repository containing the framework's source code, installation instructions, documentation, and example code is available at github.com/amitkbatra/EnergyNet.

Declarations

Conflict of interest The authors declare that they do not have any conflicts of interest to disclose.

References

1. Chen Y-H, Krishna T, Emer JS, Sze V (2017) Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J Solid-State Circuits 52(1):127–138. https://doi.org/10.1109/JSSC.2016.2616357
2. Qiu J, Wang J, Yao S, Guo K, Li B, Zhou E, Yu J, Tang T, Xu N, Song S, Wang Y, Yang H (2016) Going deeper with embedded fpga platform for convolutional neural network
3. Li L, Zhu J, Sun M-T (2019) Deep learning based method for pruning deep neural networks. In: 2019 IEEE international conference on multimedia expo workshops (ICMEW), pp 312–317 . https://doi.org/10.1109/ICMEW.2019.00-68
4. Gholami A, Kim S, Dong Z, Yao Z, Mahoney MW, Keutzer K (2021) A survey of quantization methods for efficient neural network inference. arXiv:2103.13630 [cs] [cs]
5. Pastor A, Mozo A, Vakaruk S, Canavese D, López DR, Regano L, Gómez-Canaval S, Lioy A (2020) Detection of encrypted cryptomining malware connections with machine and deep learning. IEEE Access 8:158036–158055
6. Vilalta R, Muñoz R, Casellas R, Martínez R, López V, de Dios OG, Pastor A, Katsikas GP, Monti P, Mozo A, et a(2021) Teraflow: Secured autonomic traffic management for a tera of sdn flows. In: 2021 joint European conference on networks and communications & 6G summit (EuCNC/6G Summit). IEEE, pp 377–382
7. Mozo A, Karamchandani A, de la Cal L, Gómez-Canaval S, Pastor A, Gifre L (2023) A machine-learning-based cyberattack detector for a cloud-based SDN controller. Appl Sci 13(8):4914. https://doi.org/10.3390/app13084914
8. Strubell E, Ganesh A, McCallum A(2019) Energy and policy considerations for deep learning in NLP
9. Canziani A, Paszke A, Culurciello E (2017) An analysis of deep neural network models for practical applications
10. Li D, Chen X, Becchi M, Zong Z (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), pp 477– 484 . https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.76
11. García-Martín E, Lavesson N, Grahn H, Casalicchio E, Boeva V (2019) How to measure energy consumption in machine learning algorithms. In: Alzate C, Monreale A, Assem H, Bifet A, Buda TS, Caglayan B, Drury B, García-Martín E, Gavaldà R, Koprinska I, Kramer S, Lavesson N, Madden M, Molloy I, Nicolae M-I, Sinn M (eds) ECML PKDD 2018 Workshops. Lecture notes in computer science. Springer, Cham, pp 243– 255. https://doi.org/10.1007/978-3-030-13453-2_20
12. García-Martín E, Rodrigues CF, Riley G, Grahn H (2019) Estimation of energy consumption in machine learning. J Parallel Distrib Comput 134:75–88. https://doi.org/10.1016/j.jpdc.2019.07.007
13. Patterson D, Gonzalez J, Le Q, Liang C, Munguia L-M, Rothchild D, So D, Texier M, Dean J (2021) Carbon emissions and large neural network training. arXiv:2104.10350 [cs]
14. Sharma H, Park J, Mahajan D, Amaro E, Kim JK, Shao C, Mishra A, Esmaeilzadeh H (2016) From high-level deep neural models to fpgas. In: 2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO), pp 1– 12 . https://doi.org/10.1109/MICRO.2016.7783720
15. Ma Y, Cao Y, Vrudhula S, Seo J-s (2017) An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks. In: 2017 27th international conference on field programmable logic and applications (FPL), pp 1–8 . https://doi.org/10.23919/FPL.2017.8056824
16. Rodrigues C, Riley G, Luján M (2018) Synergy: an energy measurement and prediction framework for convolutional neural networks on jetson tx1. In: PDPTA'18 - The 24th international

conference on parallel and distributed processing techniques and applications
17. Yang T-J, Chen Y-H, Sze V (2017) Designing energy-efficient convolutional neural networks using energy-aware pruning. In: 2017 IEEE conference on computer vision and pattern recognition (CVPR). IEEE, Honolulu, HI, pp 6071–6079. https://doi.org/10.1109/CVPR.2017.643
18. Cai E, Juan D-C, Stamoulis D, Marculescu D (2017) Neuralpower: predict and deploy energy-efficient convolutional neural networks. arXiv:1710.05420 [cs, stat]
19. Gordon A, Eban E, Nachum O, Chen B, Wu H, Yang T-J, Choi E (2018) Morphnet: fast & simple resource-constrained structure learning of deep networks. arXiv:1711.06798 [cs, stat]
20. Zoph B, Vasudevan V, Shlens J, Le QV (2018) Learning transferable architectures for scalable image recognition. arXiv:1707.07012 [cs, stat]
21. Liu H, Simonyan K, Yang Y(2019) Darts: differentiable architecture search. arXiv:1806.09055 [cs, stat]
22. Real E, Aggarwal A, Huang Y, Le QV (2019) Regularized evolution for image classifier architecture search. arXiv:1802.01548 [cs]
23. Jin H, Song Q, Hu X (2019) Auto-keras: an efficient neural architecture search system. arXiv:1806.10282 [cs, stat]
24. Pham H, Guan MY, Zoph B, Le QV, Dean J (2018) Efficient neural architecture search via parameter sharing. arXiv:1802.03268 [cs, stat]
25. Cai H, Zhu L, Han S (2019) Proxylessnas: direct neural architecture search on target task and hardware. arXiv:1812.00332 [cs, stat]
26. Rouhani BD, Mirhoseini A, Koushanfar F (2016) Delight: adding energy dimension to deep neural networks. In: Proceedings of the 2016 international symposium on low power electronics and design. ACM, San Francisco Airport CA USA, pp. 112–117. https://doi.org/10.1145/2934583.2934599
27. Rouhani BD, Mirhoseini A, Koushanfar F (2017) Deep3: Leveraging three levels of parallelism for efficient deep learning. In: Proceedings of the 54th annual design automation conference 2017. DAC '17. Association for Computing Machinery, New York, NY, USA, pp. 1–6. https://doi.org/10.1145/3061639.3062225
28. Stamoulis D, Cai E, Juan D-C, Marculescu D (2018) Hyperpower: Power- and memory-constrained hyper-parameter optimization for neural networks. In: 2018 Design, automation test in europe conference exhibition (DATE), pp. 19–24 . https://doi.org/10.23919/DATE.2018.8341973
29. Dai X, Zhang P, Wu B, Yin H, Sun F, Wang Y, Dukhan M, Hu Y, Wu Y, Jia Y, Vajda P, Uyttendaele M, Jha NK (2018) Chamnet: Towards efficient network design through platform-aware model adaptation. arXiv:1812.08934 [cs]
30. You Y, Gitman I, Ginsburg B (2017) Large batch training of convolutional networks. arXiv:1708.03888 [cs]
31. Lin Y, Han S, Mao H, Wang Y, Dally WJ (2020) Deep gradient compression: Reducing the communication bandwidth for distributed training. arXiv:1712.01887 [cs, stat]
32. Vogels T, Karimireddy SP, Jaggi M (2020) Powersgd: practical low-rank gradient compression for distributed optimization. arXiv:1905.13727 [cs, math, stat]
33. Tang H, Gan S, Awan AA, Rajbhandari S, Li C, Lian X, Liu J, Zhang C, He Y (2021) 1-bit adam: communication efficient large-scale training with adam's convergence speed. arXiv:2102.02888 [cs]
34. Lan Z, Chen M, Goodman S, Gimpel K, Sharma P, Soricut R (2020) Albert: a lite bert for self-supervised learning of language representations. arXiv:1909.11942 [cs]
35. Xue F, Shi Z, Wei F, Lou Y, Liu Y, You Y (2021) Go wider instead of deeper. arXiv:2107.11817 [cs]
36. Guo, Y (2018) A survey on methods and theories of quantized neural networks. *[1081] arXiv: 1808.04752 [cs, stat]
37. Zhou S, Wu Y, Ni Z, Zhou X, Wen H, Zou Y (2018) Dorefa-net: training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv:1606.06160 [cs]
38. Yang J, Shen X, Xing J, Tian X, Li H, Deng B, Huang J, Hua X (2019) Quantization networks. arXiv:1911.09464 [cs, stat]
39. Yu X, Liu T, Wang X, Tao D(2017) On compressing deep models by low rank and sparse decomposition. In: 2017 IEEE conference on computer vision and pattern recognition (CVPR), pp. 67–76 . https://doi.org/10.1109/CVPR.2017.15
40. Gou J, Yu B, Maybank SJ, Tao D (2021) Knowledge distillation: a survey. Int J Comput Vis 129(6):1789–1819. https://doi.org/10.1007/s11263-021-01453-z. arXiv:2006.05525
41. Hinton G, Vinyals O, Dean J (2015) Distilling the knowledge in a neural network. arXiv: 1503.02531 [cs, stat]
42. Sanh V, Debut L, Chaumond J, Wolf T (2020) Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs]
43. Yang G, Hu EJ, Babuschkin I, Sidor S, Liu X, Farhi D, Ryder N, Pachocki J, Chen W, Gao J (2022) Tensor programs v: Tuning large neural networks via zero-shot hyperparameter transfer. arXiv:2203.03466 [cond-mat]
44. Canziani A, Paszke A, Culurciello E (2017) An analysis of deep neural network models for practical applications. https://doi.org/10.48550/arXiv.1605.07678
45. Post-Training Quantization - TensorFlow Lite. https://www.tensorflow.org/lite/performance/post_training_quantization
46. Bergstra J, Bardenet R, Bengio Y, Kégl B (2011) Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems, vol. 24. Curran Associates, Inc.
47. Zhu M, Gupta S (2017) To prune, or not to prune: exploring the efficacy of pruning for model compression
48. Pastor A, Mozo A, Lopez DR, Folgueira J, Kapodistria A (2018) The mouseworld, a security traffic analysis lab based on nfv/sdn. In: Proceedings of the 13th international conference on availability, reliability and security, pp. 1–6
49. Mozo A, Morón-López J, Vakaruk S, Pompa-Pernía ÁG, González-Prieto Á, Aguilar JAP, Gómez-Canaval S, Ortiz JM (2022) Chlorophyll soft-sensor based on machine learning models for algal bloom predictions. Sci Rep 12(1):13529. https://doi.org/10.1038/s41598-022-17299-5
50. Vakaruk S, Sierra-García JE, Mozo A, Pastor A (2021) Forecasting automated guided vehicle malfunctioning with deep learning in a 5g-based industry 4.0 scenario. IEEE Commun Mag 59(11):102–108
51. Karamchandani A, Mozo A, Vakaruk S, Gómez-Canaval S, Sierra-García JE, Pastor A (2023) Using N-BEATS ensembles to predict automated guided vehicle deviation. Appl Intell. https://doi.org/10.1007/s10489-023-04820-0
52. Mozo A, Vakaruk S, Sierra-García JE, Pastor A (2023) Anticipatory analysis of agv trajectory in a 5g network using machine learning. J Intell Manuf:1–29
53. Vakaruk S, Karamchandani A, Sierra-García JE, Mozo A, Gómez-Canaval S, Pastor A (2023) Transformers for multi-horizon forecasting in an industry 4.0 use case. Sensors 23(7):3516