applied sciences

MDPI

Article

A Comparative Analysis for Optimizing Machine Learning Model Deployment in IoT Devices

Md. Faiyaz Abdullah Sayeedi, Jannatul Ferdous Deepti, Anas Mohammad Ishfaqul Muktadir Osmani, Taimur Rahman, Safrin Sanzida Islam and Md. Motaharul Islam *

Department of Computer Science and Engineering, United International University, Dhaka 1212, Bangladesh; msayeedi212049@bscse.uiu.ac.bd (M.F.A.S.); jdeepti212008@bscse.uiu.ac.bd (J.F.D.); aosmani203004@bscse.uiu.ac.bd (A.M.I.M.O.); trahman221427@bscse.uiu.ac.bd (T.R.); sislam212124@bscse.uiu.ac.bd (S.S.I.)
* Correspondence: motaharul@cse.uiu.ac.bd

Abstract: In the intersection of the Internet of Things (IoT) and Machine Learning (ML), the choice between high-level and low-level programming libraries presents a significant dilemma for developers, impacting not only the efficiency and effectiveness of ML models but also their environmental footprint. We have proposed a comprehensive framework to aid in this decision-making process, underpinned by a detailed comparative analysis of both types of libraries on one of the key IoT ML tasks: image classification. We have introduced a novel algorithm designed to calculate the green footprint of ML model training, factoring in execution time, memory utilization, power consumption, and CPU temperature, addressing the urgent need for sustainable ML practices. Through an empirical evaluation of popular libraries such as PyTorch for high-level and Libtorch for low-level development, we have assessed their performance, development efficiency, and hardware compatibility. The culmination of our research is a decision support system that synthesizes the experimental findings to guide developers toward choices that harmonize model performance with environmental sustainability.

Keywords: IoT device; machine learning; deep learning; energy efficiency

Citation: Sayeedi, M.F.A.; Deepti, J.F.; Osmani, A.M.I.M.; Rahman, T.; Islam, S.S.; Islam, M.M. A Comparative Analysis for Optimizing Machine Learning Model Deployment in IoT Devices. *Appl. Sci.* **2024**, *14*, 5459. https://doi.org/10.3390/app14135459

Academic Editors: Wei Liu and Chia-Huei Wu

Received: 16 May 2024
Revised: 18 June 2024
Accepted: 20 June 2024
Published: 24 June 2024

1. Introduction

In the rapidly evolving landscape of technology, the Internet of Things (IoT) has emerged as a foundational element, transforming various sectors by enabling the interconnectivity of physical devices through the Internet. These devices, ranging from simple sensors to complex systems, collect and exchange data, facilitating a smarter and more responsive environment. Concurrently, Machine Learning (ML) has seen remarkable advancements, enhancing the capability of systems to learn from data, predict outcomes, and make informed decisions without explicit programming. The integration of ML with IoT has unlocked a vast array of opportunities, automating tasks traditionally requiring human intervention and offering more insightful and predictive analytics.

The fusion of IoT with ML presents the transformative potential for developing intelligent solutions across diverse domains, including predictive maintenance, anomaly detection, and smart transportation. This integration, however, introduces a critical challenge for developers: the selection of the appropriate development approach for ML-based features within IoT solutions. The dilemma stems from a choice between leveraging traditional, high-level programming languages and libraries that simplify ML model development and opting for more specialized, low-level programming for nuanced control and customization. This decision is pivotal, impacting not only the feasibility and performance of the solution on the intended hardware but also its energy efficiency, execution time, and overall carbon footprint.

Despite the clear benefits of integrating ML into IoT-based solutions, developers face significant hurdles in selecting the most suitable development approach for ML models.

This challenge is compounded by the need to balance ease of development, performance requirements, hardware compatibility, and sustainability concerns. There is a conspicuous gap in guidance for developers navigating these complex considerations, leading to potential inefficiencies and suboptimal solutions.

To address this challenge, we have proposed the development of a decision support system designed to aid developers in selecting the most appropriate approach for developing ML models within IoT-based solutions. This system will be grounded in an exhaustive comparison of available solutions, taking into account various problem-specific and environmental factors. Through a comprehensive experimental framework, we have assessed different ML development libraries across multiple dimensions, including performance metrics, execution time, memory utilization, power consumption, CPU temperature, and carbon footprint. This approach will enable the identification of trade-offs and optimal paths for different IoT-ML application scenarios.

Our research contributes significantly to the field of IoT and Machine Learning integration via the following:

- A Comparative Analysis of ML Development Libraries: We have provided an extensive comparative analysis between high-level and low-level programming libraries for developing ML models, catering to a range of IoT machine learning tasks. This analysis leverages two distinct datasets, each aligned with a common IoT ML image classification objective.
- The Carbon Footprint Calculation Algorithm: A novel algorithm has been proposed, coined the green carbon footprint (GCF), to calculate the carbon footprint associated with the execution of ML training codes. This algorithm incorporates a comprehensive set of parameters, including execution time, memory utilization, power consumption, and CPU temperature.
- The Decision Support System: Based on the exhaustive experimental findings, a decision support system has been synthesized, presenting key insights and recommendations for developers. This system aids in navigating the complex landscape of IoT-ML integration, highlighting optimal development approaches that balance performance, hardware compatibility, and environmental sustainability.

The remainder of the paper is organized as follows. First, the existing literature related to our work and its limitations are discussed in Section 2. Then, our detailed methodology is outlined, which includes the selection of datasets and machine learning models in Section 3. Subsequently, the experimental setup configuration and evaluation metrics are outlined. The results of the experiments are interpreted in Section 4. Two case studies are then compared with the green carbon footprint algorithm that we propose in Section 5. Finally, we explore potential future research, as well as the limitations and conclusion of our work, in Section 6.

2. Related Work

In recent years, there has been much work done on determining an optimal method to train and deploy ML models on the edge device [1]. Subsequently, the selection of the programming language on the device considerably affects elements like execution speed, energy efficiency, and code complexity. This literature assessment explores the studies landscape concerning Raspberry Pi and programming languages, inspecting how language choice affects performance, complexity, and energy consumption. Early studies laid the foundation for comparing the performance of compiled and translated languages. Compiled languages, such as C and C++, exhibit better execution speeds due to direct interpretation in machine code; interpreted languages, such as Python , rely on runtime interpretation; and tasks that require more computation [2–4] perform more slowly. Furthermore, the analysis of image classification methods revealed the computational efficiency of the C# implementation on Python, albeit at a higher computational cost [3]. Beyond the compiled vs. interpreted arguments, recent research has delved into the performance differences between specific programming languages. Comparative analyses consistently

showcase the faster execution speed of C++ over Python, attributing this difference to C++'s lower-level nature [5]. Empirical comparisons between various languages emphasize scripting languages' development speed advantage, albeit with increased memory consumption [6]. Attempts to analyze energy consumption in programming languages find that compiled languages are more energy efficient than interpreted languages. Source code optimization emerges as an important factor affecting energy efficiency and performance [7]. The complexity of development is another factor to consider. Python appears as the simplest and easiest language, while C++ and Java are considered the most complex [8]. The optimization of energy efficiency and performance is important when it is a resource-constrained environment like IoT devices and microcontrollers. Edge machine learning techniques and advances in tinyML computer vision applications provide promising ways to achieve this goal [9,10]. Studies focusing on Raspberry Pi applications underscore the importance of more efficient and effective code for these devices [11–13]. Manual optimization methods are often more effective than automated compiler optimization, emphasizing the need for customized methods [14]. For performance-critical tasks, studies suggest that compiled languages like C or C++ might be preferable. Moreover, using optimization techniques and energy-efficient languages is recommended when energy consumption is a concern [15]. Performance analysis in microcontrollers reveals the capabilities of programming languages for data processing tasks. C and C++ have better performance than other languages, while MicroPython exhibits slower execution times, limiting its suitability for high-performance applications [16].

Noticeable trends in the literature emphasize the importance of resource-constrained environments such as IoT devices. Pre-trained models and edge machine learning techniques offer promising approaches to improve energy efficiency and reduce communication costs in these environments [16–18]. Furthermore, the concepts of "green coding" [19,20] and "carbon footprint" [21] emphasize the need for sustainable coding practices to reduce the environmental impact of computers. They focus on the importance of considering energy efficiency and environmental sustainability in software development [16,22,23].

Table 1 presents a gap analysis comparing existing works with the contributions of our study.

Table 1. Gap analysis of existing work.

Reference	Key Findings or Features	Identified Gaps and Our Contributions
Shoaib et al. [24]	Introduced a lightweight ML model for image classification and an optimized hardware engine for IoT devices. The image classifier was used to remove irrelavant frame from video and reduce the communication time by training the model on-device instead of relying on the cloud server. This was made possible by building the hierarchical structured accelerator tailored to be used with resource-constraint devices.	Limited to image processing tasks using one library. Focused on optimizing the hardware accelerator instead of trying out less expensive software solutions that are more versatile. The experiments can be costly to replicate due to hardware specificity. Our work extends to using low-level library, offering a more holistic IoT solution.
Benavente-Peces et al. [25]	Analyzed power consumption for ML models in cloud environments. Energy efficiency of a regular building in a smart city was measured and the building was classified according to their respective energy efficiencies using various ML models.	On-device execution was not considered. The ML algorithms used were shallow algorithms due to the structured dataset in use providing the energy efficiency of a building instead of the IoT device constructing the system. We have provided an in-depth analysis of on-device power consumption and its environmental impact on a particular device.

Table 1. *Cont.*

Reference	Key Findings or Features	Identified Gaps and Our Contributions
Stančin et al. [26]	Twenty Python libraries of six different catagories were benchmarked. Deep learning was of one of those categories that had 4 libraries. The comparision was made based on the available feature each library offered. Pytorch and Keras was recommended for rapid prototyping.	Practical evaluation was not considered to assess the efficiency of the selected libraries. Only Python libraries were analyzed instead of covering lower-level libraries written in C++. Environmental sustainability was overlooked. Our system includes an environmental impact assessment, guiding towards more sustainable ML development.
Zhang et al. [27]	Compared and analyzed the deployment of two trained model architectures using five ML libraries in multiple types of devices. The device selection ranged from low-resource to high-performing laptops.	The CPU temperature was not measured during the experiments, although it was in our case. Only trained models were loaded in the device. We train the model in the device.
Oliveira et al. [28]	Compared two commonly used deep learning libraries in Raspberry Pi by considering resource usage and energy consumption.	Limited to only high-level language library. Low-level libraries for deep learning are assessed in our study.

3. Methodology

Figure 1 depicts our proposed methodology. This section outlines the methodology employed to assess the impact of ML libraries on the carbon footprint of IoT solutions. Our approach encompasses the selection of datasets and ML models, the choice of programming libraries, and the calculation of the carbon footprint through a novel algorithm.

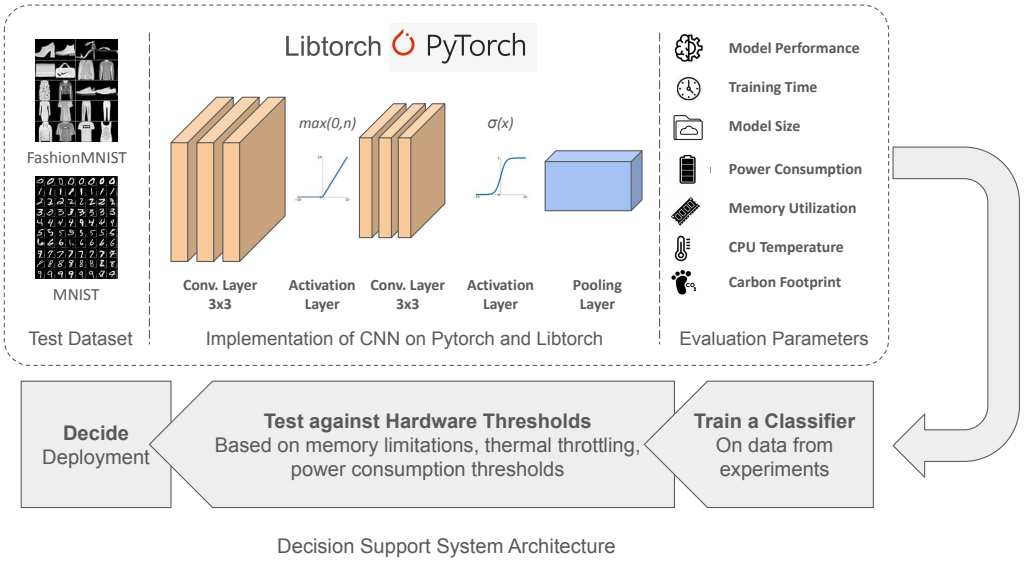

Figure 1. Proposed methodology for our experimental setup, chosen metrics, and decision support system.

3.1. Selection of Dataset and Model

In our study, we selected two standard benchmarking datasets that are symbolic of common tasks in IoT machine learning, pairing each with a model best suited to its characteristics for an insightful comparative analysis. We selected the FashionMNIST dataset (https://github.com/zalandoresearch/fashion-mnist, accessed on 15 May 2024), which comprises 60,000 grayscale images of 10 fashion categories, each of which was 28 × 28 pixels, and we utilized a Convolutional Neural Network (CNN) model. This

choice was driven by the CNN's proven capability in handling image data, particularly its effectiveness in discerning patterns and features essential for classifying different types of clothing and accessories. In addition, we also opted for the MNIST dataset (http://yann.lecun.com/exdb/mnist/, accessed on 15 May 2024), which is the predecessor of the FashionMNIST dataset. It consists of 60,000 grayscale images of the English digits (i.e., from 0 to 9) having the same resolution as the FashionMNIST. The architecture of the CNN, tailored to efficiently process the varying shapes and styles present in the FashionMNIST and the MNIST dataset, enables accurate and rapid image classification, making it an ideal match for this type of visual data common in e-commerce and fashion industry applications.

Convolutional Neural Network

Convolutional Neural Networks (CNNs) are deep learning models commonly used for computer vision. They consist of layers that can detect features using a process known as convolution, which uses activation layers to bring out more relevance to these features and pooling to reduce the dimensions for faster inference.

We have used a basic Convolutional Neural Network for both Pytorch and Libtorch to keep our experimental setup uniform. The architecture is as follows:

$$3 \times 3 \text{ Conv.} \rightarrow \text{Activation} \rightarrow 3 \times 3 \text{ Conv.} \rightarrow \text{Activation} \rightarrow \text{MaxPool} \quad (1)$$

In our study, we used basic Convolutional Neural Network (CNN) architecture for both PyTorch and Libtorch to ensure uniformity in our experimental setup. The architecture consists of two 3 × 3 convolutional layers, each followed by an activation function, and concludes with a MaxPool layer. This specific design was chosen for its balance between computational efficiency and the ability to capture essential spatial features from input images. Using 3 × 3 convolutions is a common practice that minimizes computational overhead while effectively extracting fine details.

The simplicity of this architecture facilitates the ease of implementation and reduces the error rate, which is crucial when comparing the performance of different machine learning libraries. By keeping the architecture consistent, we ensure that any observed differences in performance, power consumption, or carbon footprint can be attributed to the libraries themselves rather than variations in the network design.

The chosen architecture also provides a reliable baseline for benchmarking the performance of different machine learning libraries under controlled conditions. For the FashionMNIST dataset, which includes various patterns and textures in clothing items, this CNN architecture is particularly suitable. It captures intricate patterns with its double layer of 3 × 3 convolutions and effectively reduces spatial dimensions with the MaxPool layer, improving classification performance while maintaining computational efficiency. Additionally, the simplicity of the network helps prevent overfitting, a common issue with more complex architectures, especially when dealing with datasets of moderate complexity like FashionMNIST. This approach ensures that our comparative analysis focuses on the libraries' performance and environmental impact, providing clear insights into their capabilities and ecological footprints.

3.2. Selection of Programming Libraries

Python has primarily been the language of choice when it comes to machine learning, with libraries such as TensorFlow, Keras, PyTorch, and scikit-learn used for their ease of use and extensive support for deep learning models. With more resource-constrained devices and a shift to computation on edge devices, the overhead of the Python environment has diverted attention to lower-level languages such as C, C++, and Rust.

We have chosen Python and C++ to implement the models as mentioned earlier for comparison between libraries with regard to energy efficiency. The library selected for C++ will be Libtorch, a deep learning framework that is compatible with PyTorch, with the methods and functions being similar as well as the models built on PyTorch being able to

be run with Libtorch. The library we have selected for Python will be PyTorch; we chose this so it would be as close as possible to our Libtorch configuration.

3.3. Experimental Setup

To conduct our experiments, a Raspberry Pi 4 Model B is considered as the IoT device. Table 2 lists the specification of the specific version of the device used. A custom OS image is burnt in a 32 GB SD card and inserted in the Raspberry Pi's storage slot. For interacting with the device, we have used PuTTY (version 0.79), an open-source software that utilizes the SSH protocol to remotely access IoT devices. Experiments are conducted under controlled conditions, with each Machine Learning model trained on its respective dataset using the selected libraries. The performance metrics, execution time, memory utilization, power consumption, and CPU temperature are recorded for individual experiments.

The choice of the Raspberry Pi 4 Model B for our experiments is driven by several key factors that align with the objectives of our study. Firstly, the Raspberry Pi 4 Model B is widely recognized as a cost-effective and versatile IoT device, making it an ideal representative for real-world IoT applications. Its compact form factor and low power consumption align with the principles of sustainable computing, a core focus of our study, which aims to evaluate the environmental impact of machine learning models in IoT contexts. Moreover, its affordability ensures that the findings of our research are accessible and applicable to a broad audience, including educational institutions and industry professionals.

Table 2. Raspberry Pi 4 Model B specifications.

Entity	Specification
Processor	ARM64 Quad-Core
RAM	LPDDR4-3200 8 GB
Power consumption	2.5 A
Operating temperature	0–50 °C ambient
Interfaces	2 USB jacks, Gigabit Ethernet
Storage	32 GB micro-SD Card
Operating System	Raspberry Pi OS

3.4. Measuring Green Carbon Footprint

The Green Software Foundation's implementation of calculating carbon intensity of software [29] is defined as follows:

$$SCI = \frac{(E \times I) + M}{R} \tag{2}$$

where SCI is the Software Carbon Intensity, E is the energy consumed, I is carbon emitted per unit of energy, M is the embodied emissions of a software system, and R is the functional unit of the piece of software. This functional unit may be API calls or training runs for an ML model. Carbon emissions per kWh are a location-based parameter and can be found through reports from government agencies [30]. We have considered the number of epochs per training session as the functional unit, and the embodied emissions of the Raspberry Pi as the standard embodied emission of a smartphone.

To evaluate the environmental impact, we have introduced an additional Algorithm 1 to calculate the green footprint of training machine learning models, beyond just carbon emissions. The "Algorithm 1" considers multiple operational factors, including execution time, memory utilization, power consumption, and CPU temperature. Distinct from existing methodologies, this algorithm is designed to provide a more holistic view of the environmental impact by integrating these diverse metrics, thereby offering a comprehensive assessment of sustainability.

Algorithm 1 Calculate Carbon Footprint

Require: $executionTime, memoryUtilization, powerConsumption, cpuTemperature$
Ensure: $carbonFootprint$
$powerToCarbonConversionFactor \leftarrow$ constant
$memoryToCarbonConversionFactor \leftarrow$ constant
$powerCost \leftarrow powerConsumption \times executionTime \times powerToCarbonConversionFactor$
$memoryCost \leftarrow memoryUtilization \times memoryToCarbonConversionFactor$
$temperatureCost \leftarrow f(cpuTemperature)$ ▷ Define $f()$ based on empirical data
$carbonFootprint \leftarrow powerCost + memoryCost + temperatureCost$
return $carbonFootprint$

Furthermore, this algorithm is structured to enhance the ecological aspects of machine learning applications, making it substantially greener compared to conventional approaches. It encourages the adoption of more sustainable practices in the development and deployment of machine learning models by quantifying the carbon emissions and thereby enabling developers to make informed decisions aimed at reducing the ecological footprint.

Mathematical Representation

The green carbon footprint (GCF) is calculated as follows:

$$GCF = (P \times T \times PGCF) + (M \times MGCF) + TC \tag{3}$$

where P is the power consumption in Kilowatts (kW); T is the Execution time in hours; $PGCF$ is the Power to Carbon Conversion Factor; M is the Memory utilization in Kilobyte (kB); $MGCF$ is the Memory to Carbon Conversion Factor; and, finally, TC is the Temperature Cost, which is a function to calculate the CPU temperature.

3.5. Evaluation Metrics

The evaluation of our machine learning models encompasses a holistic approach, considering a broad spectrum of metrics to assess both traditional performance and environmental impact. This comprehensive evaluation ensures a nuanced understanding of each model's effectiveness and its sustainability within IoT applications.

Performance Metrics on Test Data: The evaluation metrics for model performance on test data include accuracy, F1-score, precision, and recall. These metrics provide a multifaceted view of the model's effectiveness.

- Accuracy: Accuracy is the proportion of correctly predicted observations out of the total observations. It provides a general measure of the model's performance. However, in scenarios with imbalanced datasets, accuracy might not be sufficient to understand the model's true performance. In our context, accuracy helps to obtain the overall correctness of our ML models across various IoT tasks.
- Precision: The ratio of correctly predicted positive observations to the total predicted positive observations. It reflects the model's ability to return relevant results. Precision is crucial in IoT-based real-world applications, where false positives can lead to unnecessary negative consequences that can be costly.
- Recall (Sensitivity): The ratio of correctly predicted positive observations to all observations of individual classes. It measures the model's capability to find all relevant cases. High recall is particularly important in scenarios where missing positive instances can lead to significant issues, such as in security or health monitoring IoT applications.
- F1 Score: The harmonic mean of precision and recall. This score takes both false positives and false negatives into account, making it a better measure of the incorrectly classified cases than the accuracy metric alone. The F1 Score is critical in our study as it offers a more summarized measure of the model's performance by accounting for both

false positives and false negatives, thus providing a comprehensive understanding of the model in various IoT applications.

These performance metrics offer a comprehensive assessment of the model's predictive accuracy and its ability to manage trade-offs between precision and recall, essential in applications where the cost of false positives differs from that of false negatives.

Computational and Environmental Impact Metrics: Additionally, the evaluation includes metrics such as execution time, memory utilization, power consumption, and CPU temperature. These metrics are pivotal for assessing the model's computational efficiency, resource demands, and environmental impact, guiding towards sustainable ML deployment in IoT systems. When applicable, we have used command line utilities to remain programming-language-agnostic.

- Execution Time: Measures the total time taken by the machine learning model to complete its training process. A shorter execution time is generally preferable, indicating an efficient model capable of learning quickly, which is crucial in time-sensitive or resource-limited environments. In Python, the execution time can be accurately measured using the `time` library.
- Memory Utilization: Reflects the amount of memory required by the machine learning models during their execution. Effective memory management is critical, especially in environments with limited memory resources such as IoT devices. The command line utility `free` is used to monitor and optimize memory consumption during model training.
- Power Consumption: Indicates the electrical power used by the computing device while running the machine learning model. Energy efficiency is paramount in scenarios where power availability is constrained, such as mobile or embedded systems. Power consumption is typically measured using a USB power meter or multimeter to track the voltage and current draw of a device, like a Raspberry Pi, thus allowing for the calculation of power usage in watts. Power draw can be estimated using `powertop`. The command also logs CPU usage.
- CPU Temperature: Monitoring CPU temperature is essential for maintaining system stability and efficiency. High temperatures may suggest excessive computational demand or inadequate cooling, potentially leading to hardware throttling or failure. Temperature can be tracked using the `vcgencmd` utility on the device, which allows for real-time temperature monitoring. This will allow us critical data for managing thermal conditions during intensive computational tasks.

3.6. Decision Support System Framework

Based on the experimental results, we have proposed a theoretical decision support system for the developers. It can aid in choosing a suitable approach for addressing a specific problem set. First of all, we have taken the type of data associated with the target scenario under consideration, which can be image data or other types of data. Then, our focus is shifted to the minimum threshold for certain parameters that are dependent on the problem at hand. Some of the parameters are hardware-specific like CPU utilization, CPU, and memory consumption. Other parameters include model-specific parameters like model accuracy, precision, recall, and F1-Score.

This structured methodology provides a comprehensive framework for assessing the environmental impact of ML models in IoT applications, offering insights into the trade-offs between ease of development, performance, and sustainability.

The pipeline in the decision support system follows a sequential process to handle data from its raw form to the deployment of a machine learning model. This process is detailed below:

1. Raw Data Collection: Gathering unprocessed data from various sources relevant to the application domain.
2. Data Transformation: Applying transformations to the raw data to convert them into a more manageable or suitable format for analysis.
3. Data Normalization: Normalizing data to ensure that the model is not biased towards variables with larger scales.
4. Data Cleaning: Removing or correcting data anomalies, missing values, and outliers to improve data quality.
5. Data Compression: Reducing the size of the data through various compression techniques to enhance storage efficiency and speed up processing.
6. Feeding Data to Model: Inputting the processed data into the machine learning model for training.
7. Model Deployment: Deploying the trained model into a real-world environment where it can provide insights or make decisions based on new data.

This comprehensive pipeline not only optimizes the data handling and model training processes but also ensures that the deployed models are robust, efficient, and suitable for the intended IoT applications. Figure 2 shows the architecture for our decision support system.

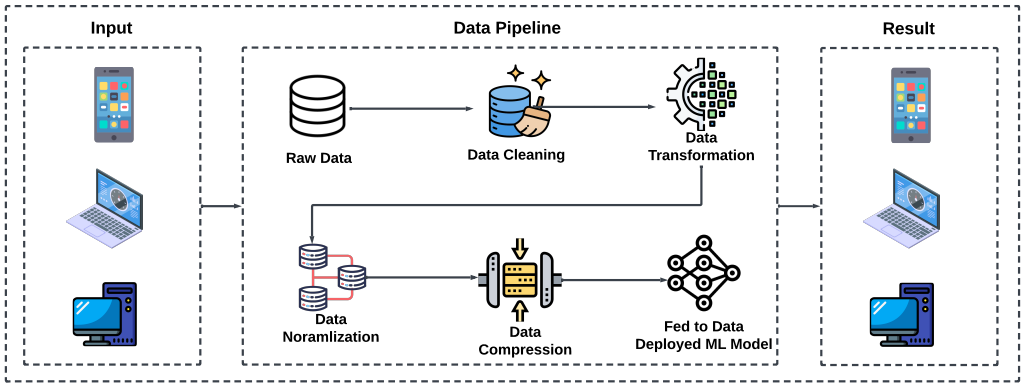

Figure 2. Decision support system architecture.

To illustrate the practical application of our decision support system (DSS), we present detailed case studies involving real-world scenarios. For instance, consider an IoT-based smart agriculture system where image data are utilized to monitor crop health. In this case, the DSS guides the developer through selecting an appropriate machine learning model that balances high accuracy in disease detection with minimal energy consumption, ensuring sustainability in a resource-constrained environment. Another example is a smart healthcare system using wearable devices to collect and analyze patient data. Here, the DSS helps in choosing models that not only achieve high precision in anomaly detection but also operate efficiently on low-power devices, enhancing both patient safety and device longevity. These case studies demonstrate the DSS's ability to provide solutions that align with specific application requirements, emphasizing the trade-offs between performance, ease of development, and environmental impact. This practical implementation underscores the DSS's potential to facilitate the development of sustainable and efficient IoT applications across various domains. Figure 3 shows the simple User Interface (UI) Design for our decision support system.

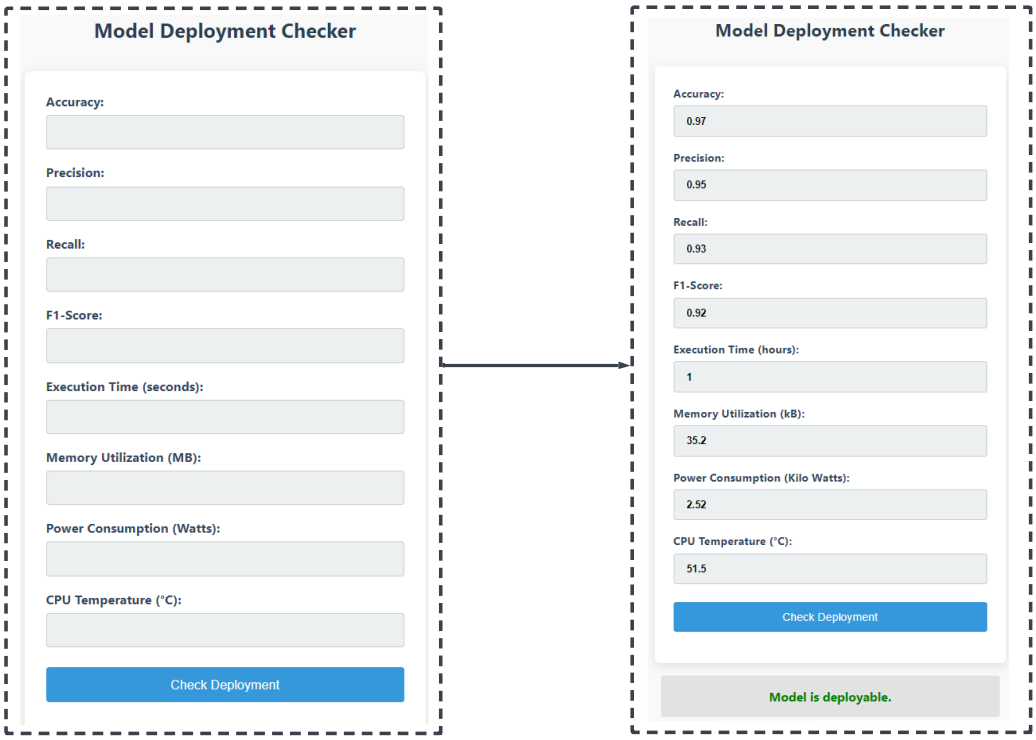

Figure 3. User Interface (UI) Design for the decision support system.

4. Result Analysis

In this section, we have analyzed our findings from two distinct perspectives. First of all, the model's performance and report of the traditional evaluation metrics, namely, accuracy, precision, recall, and F1-Score, is presented. We have compared these metrics between the ML libraries in our experiment. Secondly, the various metrics associated with the computational and environmental impact of the model like resource consumption and the model's carbon footprint are measured. All of the model training is carried out for 10 epochs to conduct a fair comparison. In Section 5, we ultimately validate our new green carbon footprint (GCF) algorithm through a case study.

4.1. Training on FashionMNIST

Table 3 lists the traditional performance metrics for training an image classifier on the FashionMNIST dataset using the PyTorch and Libtorch libraries.

Table 3. Model performance on FashionMNIST dataset. The best results are given in bold.

Metrics	PyTorch	Libtorch
Accuracy	89.52%	**89.7%**
Precision	**89.50%**	88.7%
Recall	**89.52%**	89.3%
F1-Score	**89.48%**	88.9%
Training time	3.03 h	**2.01 h**
Total Power	2.925 kW	**2.511 kW**
Average Temperature	51.4 °C	**50.2 °C**

Analyzing the performance of PyTorch and Libtorch on the FashionMNIST dataset reveals distinct advantages for each framework. Both PyTorch and Libtorch performed in a similar manner in terms of the model's prediction capability. Nonetheless, Libtorch had slightly better results while taking a shorter time to train the model.

Moreover, Libtorch showcases advantages in training efficiency and resource management. It completes training faster by approximately 7% and utilizes less memory and power, which could be crucial in resource-constrained or energy-sensitive environments. Additionally, Libtorch operates at a slightly lower CPU temperature, suggesting more efficient resource use and, potentially, a lower environmental impact. Further comparisons may be drawn with Figure 4, where Libtorch has far more optimized and uniform memory utilization and Pytorch is slightly more sporadic in CPU calls over time.

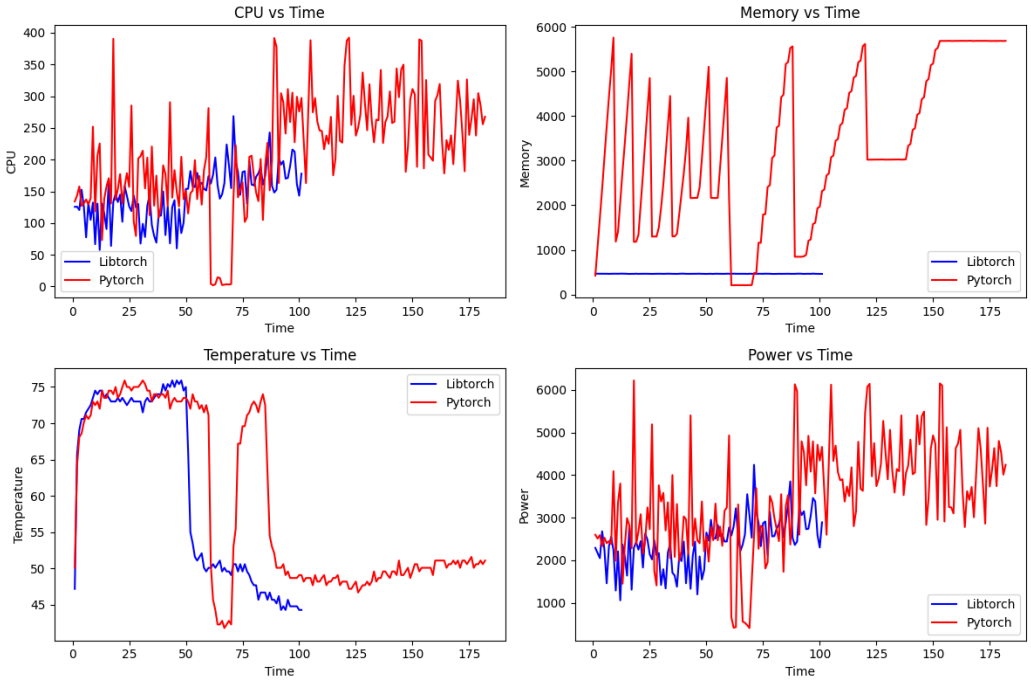

Figure 4. Comparison graph plots for FashionMNIST dataset.

Figure 4 depicts the continuous tracking of the external factors for the duration of the training of the model on the FashionMNIST dataset. The metrics are reported every 60 s in a CSV file and then plotted using the `matplotlib` library in Python. Firstly, by comparing the CPU usage it is apparent that PyTorch uses more CPU computation resources consistently while Libtorch's CPU usage remains low for the first half of the training period and shows a slight increase for the rest of the training period, which is still lower than that of PyTorch. In terms of the memory consumption profile, there is a drastic difference between the two libraries. The PyTorch implementation shows a sustaining fluctuation and an overall higher memory consumption. In comparison, the Litorch model training shows an almost linear consumption of memory that flattens at around 460 megabytes. For the internal temperature of the CPU, PyTorch surprisingly is lower for most of the time except for a sharp increase right after the initial execution of the training session. Meanwhile, the CPU temperature stays around 80 degrees Celsius for the half of the training period and then significantly decreases to the ambient operating temperature of the device. Finally, the power consumption profile for both PyTorch and Libtorch shows a direct correlation with the respective CPU utilization profiles for each library.

4.2. Training on MNIST Dataset

Table 4 lists the traditional performance metrics for training an image classifier on the MNIST dataset using the PyTorch and Libtorch libraries.

Table 4. Model performance on MNIST dataset. The best results are given in bold.

Metrics	PyTorch	Libtorch
Accuracy	98.53%	**98.80%**
Precision	98.52%	**98.73%**
Recall	98.51%	**98.79%**
F1-Score	98.51%	**98.75%**
Training time	**1.20 h**	1.29 h
Total Power	0.21 kW	**0.14 kW**
Average Temperature	**76.3** °C	78.5 °C

Training on the MNIST dataset using PyTorch and Libtorch takes less time compared to the FashionMNIST experiments, mainly because of the simplicity of this dataset. In terms of the model's performance, Libtorch outperforms PyTorch by a small magnitude. Furthermore, the device memory consumption and average temperature during the training are lower for Libtorch. However, Libtorch requires slightly longer to train.

In summary, while PyTorch provides an easier approach for development as it abstracts most of the inner logic from the developers, Libtorch may be preferable for scenarios prioritizing more reliable performance and sustainability due to its lower resource consumption and energy usage. As evidenced by Figure 5, the running code on Python significantly increases CPU and RAM utilization, directly affecting the power draw in the process. A smaller power footprint despite the longer duration of training means there may be more avenues for lowering the carbon footprint. Nevertheless, Libtorch implementation is not as widely adapted as Pytorch and requires the handling of more complex coding compared to PyTorch. The choice between these frameworks would depend on the specific trade-offs between performance accuracy and operational efficiency relevant to the deployment context.

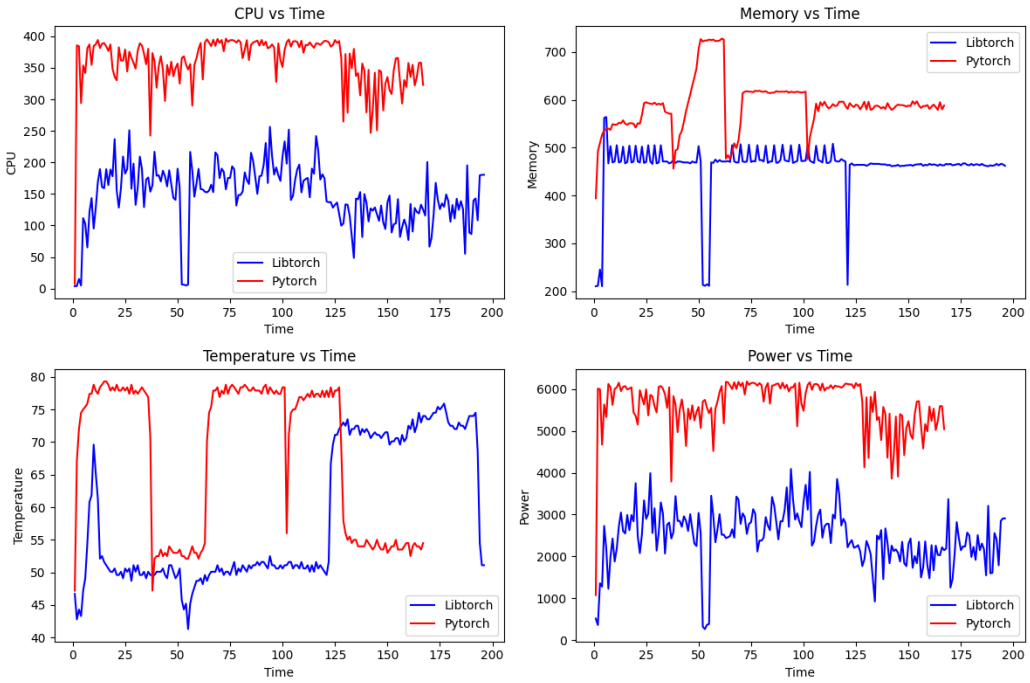

Figure 5. Comparison graph plots for MNIST dataset.

Figure 5 highlights the differences in external metrics over time during model training on the MNIST dataset. Time is measured in minutes, and CPU utilization is based on the number of calls at that timeframe. We can see a more uniform distribution for PyTorch but overall lower peaks for LibTorch. Memory is measured in megabytes, whereas similarly, LibTorch uses less memory compared to PyTorch. This is due to the overhead from the Python environment. Both temperature curves peak at around 80 °C, beyond the normal operating limit for the Raspberry Pi 4. Power consumption, measured in milliwatts, is directly related to and estimated from CPU calls; therefore, we see a smaller overall footprint from Figure 5.

4.3. Validation of the Green Carbon Footprint Algorithm

In developing the green carbon footprint calculation algorithm, one of the critical steps involves fine-tuning the conversion factors that translate the operational metrics (power consumption, memory utilization, and CPU temperature) into their equivalent carbon emissions. This process is essential not only for achieving accuracy in the calculated footprint but also for ensuring that the algorithm reflects true environmental impacts, allowing for effective sustainability assessments in machine learning operations.

4.3.1. Determining Conversion Factors

The conversion factors, namely, the Power to Carbon Conversion Factor (PGCF), Memory to Carbon Conversion Factor (MGCF), and a factor for the Temperature Cost (based on CPU temperature), are derived from a combination of industry standards, published research, and empirical data specific to the computational resources being assessed. This multi-source approach ensures a robust basis for the conversions, reflecting variations in energy sources, the efficiency of memory use, and the thermal characteristics of computing equipment.

- Power to Carbon Conversion Factor (PGCF): This factor converts the energy consumed (in kWh) by the system into grams of a CO2 equivalent. It is influenced by the regional energy mix and the efficiency of the power supply. To establish this factor, data from local utility companies and renewable energy contributions are considered.
- Memory to Carbon Conversion Factor (MGCF): Memory utilization is converted to carbon emissions based on the energy cost of memory operations and the production impact of the memory hardware. This factor accounts for the embodied carbon of producing and maintaining memory hardware, amortized over its operational lifespan.
- Temperature Cost Conversion: The CPU temperature directly affects the system's energy efficiency and cooling requirements. A higher temperature indicates more energy-intensive cooling, translating to higher emissions. A mathematical model is developed to relate CPU temperature to carbon output, using empirical data from thermal performance studies.

4.3.2. Empirical Validation and Adjustments

To validate these factors, a series of controlled experiments and simulations are conducted. These tests involve running typical machine learning models under varied conditions to capture real-time data on power consumption, memory usage, and CPU temperatures. The resulting emissions are then calculated using the initial set of conversion factors.

- Iterative Adjustments: The initial results are compared against independently verified benchmarks of carbon emissions for similar operations [31]. Discrepancies lead to iterative adjustments in the conversion factors. This iterative process continues until the calculated carbon emissions align closely with the benchmark data, ensuring accuracy and reliability.

- Algorithm Sensitivity Analysis: Sensitivity analysis is also performed to understand how changes in each factor affect the overall carbon footprint. This analysis helps prioritize adjustments in the conversion factors to minimize errors in the footprint estimation.

4.3.3. Finalization and Implementation

Once the conversion factors are optimized and validated, they are finalized and implemented into the algorithm. Regular updates and recalibrations are scheduled based on new data and changes in technology or energy metrics, ensuring that the algorithm remains accurate over time and reflective of current conditions.

This detailed approach to optimizing conversion factors underpins the reliability and ecological sensitivity of the green carbon footprint calculation algorithm, making it a vital tool for assessing the environmental impact of machine learning operations.

Using the image classification performance metrics obtained from the PyTorch and Libtorch libraries, we have validated the green carbon footprint (GCF) algorithm. For this validation, we have employed the recorded training time, model size, and total power consumption, alongside a predefined Temperature Cost based on CPU temperature.

5. Case Studies

The validity of our algorithm from Section 3.4 can be analyzed from our experimental setup. We use the data collected from Tables 3 and 4 to compare the footprint of PyTorch against LibTorch. We have used the example of both PyTorch and LibTorch trained on FashionMNIST and MNIST to show the calculations. The model sizes will be used in our calculations and are as follows:

- PyTorch Model Size: 34.5 Kb
- LibTorch Model Size: 33.7 Kb

5.1. Case Study 1: FashionMNIST with PyTorch

Using a Convolutional Neural Network model trained on FashionMNIST based on our results from PyTorch in Table 3 and Figure 4, the GCF algorithm can be formulated. We have first explored the Temperature Cost calculation and the carbon factor adjustment for memory and power. These values have been adjusted based on the Raspberry Pi 4 and importance of factors.

5.1.1. Temperature Cost Calculation

Given the CPU temperature of 51.4 °C, the Temperature Cost (TC) is calculated using a preliminary empirical formula:

$$TC = 0.1 \times 51.4 = 5.14$$

The weight of 0.1 can be adjusted based on the importance of a low thermal profile.

5.1.2. Factor Adjustment

Initial conversion factors were adjusted to align the GCF within the target range of 6 to 8 g of CO_2 equivalent. This range is calculated using the standard formula for calculating carbon footprints [31], which is commonly employed in environmental impact assessments to ensure consistency with internationally recognized emission estimation methodologies. The factors set are:

$$PGCF = 0.42, \quad MGCF = 0.06$$

These factors were found to work with the Raspberry Pi 4, and further adjustment will be required for other hardware.

5.1.3. GCF Calculation

Using the adjusted factors, the GCF for the training process is calculated as follows:

$$\begin{aligned} GCF &= (P \times T \times PGCF) + (M \times MGCF) + TC \\ &= (2.925 \times 3.03 \times 0.42) + (34.5 \times 0.06) + 5.14 \\ &= (8.86275 \times 0.42) + (2.07) + 5.14 \\ &= 3.72235 + 2.07 + 5.14 \\ &= 10.9323 \text{ units} \end{aligned}$$

Since the resulting value of 10.9323 units is near our target, it confirms the algorithm's environmental sensitivity and precision.

5.2. Case Study 2: FashionMNIST with LibTorch

The same architecture for a Convolutional Neural Network was used within LibTorch, the underlying binary distribution for PyTorch. Being implemented in a low-level language like C++, a smaller footprint can be observed, as evidenced by the values from Table 3 and Figure 4. Due to Python's reliance on a virtual environment and C++ being more inline with device archtiecture, we have seen a relatively significant difference in training time and power consumption. Furthermore, the temperature and model size on LibTorch were lower, so a smaller GCF value should be expected.

We can similarly find the factors for LibTorch, using the same conversion factors from Section 5.1.2.

$$TC = 0.1 \times 50.1 = 5.01$$

The GCF calculation will be as follows:

$$\begin{aligned} GCF &= (P \times T \times PGCF) + (M \times MGCF) + TC \\ &= (2.511 \times 2.01 \times 0.42) + (33.7 \times 0.06) + 5.01 \\ &= 2.1198 + 2.022 + 5.01 \\ &= 9.1517 \text{ units} \end{aligned}$$

Repeating these steps for the LibTorch model trained on FashionMNIST, we obtain a resulting GCF of 9.1517 units. Since the resulting GCF of 10.932355 units is higher than the corresponding value from LibTorch, it has a lower carbon footprint. Given the factors of temperature, time taken, and memory space being lower, the GCF algorithm can give an overall idea of the footprint of a machine learning model.

5.3. Case Study 3: MNIST with PyTorch

In order to find the GCF for MNIST trained on PyTorch, we can use identical parameters from the previous two case studies using our values from Table 4.

The GCF calculation will be as follows:

$$\begin{aligned} GCF &= (P \times T \times PGCF) + (M \times MGCF) + TC \\ &= (0.21 \times 1.20 \times 0.42) + (34.5 \times 0.06) + 7.63 \\ &= 0.106 + 2.07 + 7.63 \\ &= 9.806 \text{ units} \end{aligned}$$

5.4. Case Study 4: MNIST with LibTorch

Similarly, if we find the GCF for MNIST trained on LibTorch, we can use identical parameters from the previous two case studies using our values from Table 4

The GCF calculation will be as follows:

$$\begin{aligned} GCF &= (P \times T \times PGCF) + (M \times MGCF) + TC \\ &= (0.14 \times 1.29 \times 0.42) + (33.7 \times 0.06) + 7.85 \\ &= 0.07585 + 2.022 + 7.85 \\ &= 9.948 \text{ units} \end{aligned}$$

Comparing our results from Case Studies 3 and 4, we see PyTorch marginally being better than LibTorch in this scenario. The major contributing factor to this comes from the Temperature Cost, which is a lot higher on LibTorch.

This validation confirms the practical application of the green carbon footprint algorithm, underlining its sensitivity and precision in reflecting real-world operational carbon footprints of ML training processes. Continuous adjustments and validations are essential for maintaining the algorithm's alignment with evolving environmental standards and operational practices.

6. Conclusions

In conclusion, our research provides a comprehensive analysis of the environmental implications of selecting between high-level and low-level programming libraries in IoT machine learning implementations. While the high-level programming library has proven to be more developer-friendly by abstracting most of the inner logic that is close to hardware, the low-level libraries allow for more custom building of the compiled program for training the machine learning model. Hence, both approaches require sacrificing either the time required to establish if a low-level library is used or the environmental surplus if the high-level library is used. By developing a novel algorithm to quantify the carbon footprint of training ML models, we have offered a practical tool for developers to make more informed decisions that prioritize sustainability alongside performance and ease of use. The comparative study between libraries like PyTorch and Libtorch sheds light on the nuanced trade-offs involved in library selection, influencing both computational efficiency and environmental impact.

Our future work will focus on expanding the scope of our decision support system to include additional programming libraries and platforms, thereby enhancing its utility and relevance. In addition, conducting experiments with other types of data like text, audio, and structured data as used in traditional machine learning may provide further insights regarding the utility of shifting towards low-level ML libraries. We have also planned to integrate real-time data tracking and analysis features to dynamically assess the environmental impact of ML operations. This could enable more adaptive and responsive ML development practices. Furthermore, there are other machine learning libraries that can be used for comparisons. On the higher end, Python has libraries in TensorFlow (particularly TensorFlow-Lite) that have similar possibilities for model training, and on the lower end, there are libraries such as Candle developed by HuggingFace. Exploring more machine learning libraries on resource-constrained devices is another avenue for future work. By continually updating our framework to reflect the latest advancements in technology and sustainability metrics, our aim is to contribute significantly to the reduction in the ecological footprint of machine learning globally.

The limitations of this work are as follows: Raspberry Pi 4 is a widely supported device, but it may not be indicative of all of the scenarios of deployment of deep learning models. Other resource-constrained devices such as the Jetson Nano and Arduino Nano BLE are lower-profile compared to the Raspberry Pi, and finding limitations on those devices may yield fruitful results. Furthermore, the deployed settings may have bottlenecks in communication and data transmission, not from CPU usage or time taken in training. Research in a realistic network setting is another limitation of this work.

Author Contributions: Conceptualization, M.F.A.S., A.M.I.M.O. and T.R.; Methodology, M.F.A.S., A.M.I.M.O. and T.R.; Software, A.M.I.M.O. and T.R.; Validation, M.F.A.S., A.M.I.M.O. and T.R.; Formal analysis, M.F.A.S., A.M.I.M.O. and T.R.; Investigation, J.F.D. and S.S.I.; Writing—original draft, M.F.A.S., J.F.D., A.M.I.M.O. and T.R.; Writing—review & editing, M.M.I.; Supervision, M.M.I.; Project administration, M.M.I.; Funding acquisition, M.M.I. All authors have read and agreed to the published version of the manuscript.

Funding: Funded by the Institute for Advanced Research Publication Grant of United International University, Ref. No.: IAR-2024-Pub-032.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fanariotis, A.; Orphanoudakis, T.; Kotrotsios, K.; Fotopoulos, V.; Keramidas, G.; Karkazis, P. Power efficient machine learning models deployment on edge IoT devices. *Sensors* **2023**, *23*, 1595. [CrossRef] [PubMed]
2. Rysak, P. Comparative analysis of code execution time by C and Python based on selected algorithms. *J. Comput. Sci. Inst.* **2023**, *26*, 93–99. [CrossRef]
3. Salihu, B.; Tafa, Z. On Computational Performances of the Actual Image Classification Methods in C# and Python. In Proceedings of the 2020 9th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 8–11 June 2020; pp. 1–5.
4. de la Fraga, L.G.; Tlelo-Cuautle, E.; Azucena, A.D.P. On the execution time of a computational intensive application in scripting languages. In Proceedings of the 2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT), Merida, Mexico, 25–27 October 2017; pp. 149–152.
5. Zehra, F.; Javed, M.; Khan, D.; Pasha, M. Comparative Analysis of C++ and Python in Terms of Memory and Time. *Preprints* **2020**. [CrossRef]
6. Prechelt, L. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Comput.* **2000**, *33*, 23–29. [CrossRef]
7. Georgiou, S.; Kechagia, M.; Spinellis, D. Analyzing programming languages' energy consumption: An empirical study. In Proceedings of the 21st Pan-Hellenic Conference on Informatics, Larissa, Greece, 28–30 September 2017; pp. 1–6.
8. Onyango, K.A.; Mariga, G.W. Comparative Analysis on the Evaluation of the Complexity of C, C++, Java, PHP and Python Programming Languages based on Halstead Software Science. *Int. J. Comput. Inf. Technol.* **2023**, *12*, 8–19.
9. Mengistu, D.; Frisk, F. Edge machine learning for energy efficiency of resource constrained IoT devices. In Proceedings of the SPWID 2019: The Fifth International Conference on Smart Portable, Wearable, Implantable and Disabilityoriented Devices and Systems, Nice, France, 28 July–2 August 2019; pp. 9–14.
10. Lu, Q.; Murmann, B. Enhancing the energy efficiency and robustness of TinyML computer vision using coarsely-quantized log-gradient input images. *ACM Trans. Embed. Comput. Syst.* **2023**, *23*, 1–20. [CrossRef]
11. Blum, R.; Bresnahan, C. *Python Programming for Raspberry Pi*; Pearson Education: Upper Saddle River, NJ, USA, 2013.
12. Kesrouani, K.; Kanso, H.; Noureddine, A. A preliminary study of the energy impact of software in raspberry pi devices. In Proceedings of the 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Online, 10–13 September 2020; pp. 231–234.
13. Ghael, H.D.; Solanki, L.; Sahu, G. A review paper on raspberry pi and its applications. *Int. J. Adv. Eng. Manag. (IJAEM)* **2020**, *2*, 4.
14. Corral-García, J.; González-Sánchez, J.L.; Pérez-Toledano, M.Á. Evaluation of strategies for the development of efficient code for Raspberry Pi devices. *Sensors* **2018**, *18*, 4066. [CrossRef] [PubMed]
15. Abdulkareem, S.A.; Abboud, A.J. Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics). *Iop Conf. Ser. Mater. Sci. Eng.* **2021**, *1076*, 012046. [CrossRef]
16. Plauska, I.; Liutkevičius, A.; Janavičiūtė, A. Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller. *Electronics* **2022**, *12*, 143. [CrossRef]
17. Besimi, N.; Çiço, B.; Besimi, A.; Shehu, V. Using distributed raspberry PIs to enable low-cost energy-efficient machine learning algorithms for scientific articles recommendation. *Microprocess. Microsyst.* **2020**, *78*, 103252. [CrossRef]
18. Mehmood, M.U.; Chun, D.; Han, H.; Jeon, G.; Chen, K.; et al. A review of the applications of artificial intelligence and big data to buildings for energy-efficiency and a comfortable indoor living environment. *Energy Build.* **2019**, *202*, 109383. [CrossRef]
19. Lannelongue, L.; Grealey, J.; Inouye, M. Green algorithms: Quantifying the carbon footprint of computation. *Adv. Sci.* **2021**, *8*, 2100707. [CrossRef] [PubMed]
20. Herelius, S. Green Coding: Can We Make Our Carbon Footprint Smaller through Coding? Bachelor's Thesis, Blekinge Institute of Technology, Karlskrona, Sweden, 2022.
21. Pandey, D.; Agrawal, M.; Pandey, J.S. Carbon footprint: Current methods of estimation. *Environ. Monit. Assess.* **2011**, *178*, 135–160. [CrossRef] [PubMed]

22. Vieira, G.; Barbosa, J.; Leitão, P.; Sakurada, L. Low-cost industrial controller based on the raspberry pi platform. In Proceedings of the 2020 IEEE International Conference on Industrial Technology (ICIT), Buenos Aires, Argentina, 26–28 February 2020; pp. 292–297.
23. Abdulsalam, S.; Lakomski, D.; Gu, Q.; Jin, T.; Zong, Z. Program energy efficiency: The impact of language, compiler and implementation choices. In Proceedings of the International Green Computing Conference, Dallas, TX, USA, 3–5 November 2014; pp. 1–6.
24. Shoaib, M.; Venkataramani, S.; Hua, X.S.; Liu, J.; Li, J. Exploiting on-device image classification for energy efficiency in ambient-aware systems. In *Mobile Cloud Visual Media Computing: From Interaction to Service*; Springer: Cham, Switerland, 2015; pp. 167–199.
25. Benavente-Peces, C.; Ibadah, N. Buildings energy efficiency analysis and classification using various machine learning technique classifiers. *Energies* **2020**, *13*, 3497. [CrossRef]
26. Stančin, I.; Jović, A. An overview and comparison of free Python libraries for data mining and big data analysis. In Proceedings of the 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 20–24 May 2019; pp. 977–982.
27. Zhang, X.; Wang, Y.; Shi, W. {pCAMP}: Performance Comparison of Machine Learning Packages on the Edges. In Proceedings of the USENIX Eorkshop on Hot Topics in Edge Computing (HotEdge 18), Boston, MA, USA, 11–13 July 2018.
28. Oliveira, L.P.; Santos, J.H.d.S.; de Almeida, E.L.; Barbosa, J.R.; da Silva, A.W.; de Azevedo, L.P.; da Silva, M.V. Deep learning library performance analysis on raspberry (IoT device). In *Proceedings of the International Conference on Advanced Information Networking and Applications*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 383–392.
29. Green Software Foundation. *Software Carbon Intensity Standard*; Green Software Foundation: San Francisco, CA, USA, 2021.
30. Climate Transparency. Climate Transparency Report 2022. 2022. Available online: https://www.climate-transparency.org/g20-climate-performance/g20report2022 (accessed on 15 May 2024).
31. Dodge, J.; Prewitt, T.; Tachet des Combes, R.; Odmark, E.; Schwartz, R.; Strubell, E.; Luccioni, A.S.; Smith, N.A.; DeCario, N.; Buchanan, W. Measuring the carbon intensity of ai in cloud instances. In Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency, Seoul, Republic of Korea, 21–24 June 2022; pp. 1877–1894.