Hardware-Aware Energy Estimator pipeline for μNAS

Martynas Šmitas[1]
Supervisor(s): Hao Liu[1], **Marco Zuñiga Zamalloa**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Martynas Šmitas
Final project course: CSE3000 Research Project
Thesis committee: Hao Liu, Marco Zuñiga Zamalloa, Jana Weber

Abstract

Running deep learning models directly on Microcontroller Units (MCUs)—a field known as TinyML—enables artificial intelligence in energy-restricted applications that rely strictly on limited battery power or harvested local energy. Creating models for these constrained devices requires strict optimization via Neural Architecture Search (NAS) frameworks like μNAS. However, traditional proxies like Multiply-Accumulate (MAC) counts fail to serve as accurate energy predictors because they overlook complex hardware interactions and compiler-level runtime optimizations introduced by deployment engines like TensorFlow Lite (TFLite) Micro.

In this paper, we explore accurate energy estimation within the μNAS framework. We build an automated Hardware-in-the-Loop (HIL) profiling pipeline to deploy diverse architectures on an MCU and record their physical power draw, generating a dataset of 671 unique models. Evaluating a baseline linear regression predictor applied to MAC counts achieves a high macro-level fit ($R^2 = 0.985$) but suffers from an unacceptable Mean Absolute Percentage Error (MAPE) of 85.4% due to its structural oversights.

To address this limitation, we propose a novel energy estimator based on a Directed Acyclic Graph Neural Network (DAGNN). By processing the neural network topology directly, the DAGNN effectively learns complex hardware interactions and runtime optimization behaviors. Our estimator dramatically outperforms the baseline, reducing the MAPE from 85.4% down to 16.0%.

1 Introduction

Microcontroller Units (MCUs) are tiny, single-chip computers that quietly run the electronics around us, from wearable devices that recognize human activity from on-body motion sensors [1] to always-listening keyword-spotting devices [2]. Many of these systems are deployed where running a power cable is impractical: a collar that classifies a grazing animal's gait directly on its onboard microcontroller [3], a bridge-mounted sensor that detects structural damage from on-device vibration analysis [4], or a smart-irrigation node that runs its prediction model entirely on a battery-powered STM32 microcontroller in the field [5]. Such devices cannot rely on a constant power grid; instead, they must operate strictly on limited battery power or locally harvested energy, such as solar power [6]. Running deep learning models directly on these devices—a field known as TinyML—gives them the ability to interpret sensor data on-device, instead of transmitting raw data to the cloud for processing [7].

Designing a deep learning model for an MCU is considerably harder than designing one for a smartphone or server-grade GPU. MCUs typically have only a few hundred kilobytes of volatile memory (RAM) and a few megabytes of non-volatile storage, and each inference must complete within a strict latency budget [8]. A model that is too large simply does not fit on the device; one that is too slow drains the battery before it can deliver a useful number of inferences. Manually tuning an architecture to satisfy these memory, storage, and latency constraints while preserving accuracy is a slow, trial-and-error process [8].

Neural Architecture Search (NAS) automates this trade-off. NAS frameworks such as μNAS [8] search a large space of candidate architectures and select those that maximize accuracy subject to the hardware's memory, storage, and latency limits, removing much of the manual tuning burden from the model designer. For energy-restricted applications, however, fitting within these limits is not enough: the device's energy budget, set by its battery capacity or its rate of energy harvesting, must also be respected, motivating dedicated energy-aware NAS frameworks [9]. To optimize deployment, μNAS uses Multiply-Accumulate (MAC) operation counts as a proxy for hardware cost; yet such coarse proxies are known to predict both latency and energy poorly on MCUs, since they ignore instruction-level effects like scheduling, caching, and compiler optimizations [10]. Dedicated hardware-aware predictors that avoid this pitfall have, until recently, predominantly targeted more powerful platforms such as mobile processors and GPUs [11, 12]; only very recent instruction-level approaches have begun to close this gap for MCUs [10], though integrating such fine-grained estimation directly into a NAS search loop like μNAS's remains unexplored. Therefore, this work investigates whether we can integrate a more accurate, hardware-specific energy estimator directly into μNAS.

Developing this estimator presented two primary challenges. First, the energy consumption of a given MCU (like the nRF52840 that lives on the Arduino Nano 33 BLE) can vary significantly based on its configuration—such as whether Bluetooth is enabled—and the presence of other board components like external memory or voltage dividers. Therefore, to accurately predict energy usage on these low-power devices, we must measure their actual power draw. To avoid the tedious process of recording this data manually, we developed an Automated Energy Profiling Pipeline. Second, on-device deployment relies on TensorFlow Lite for Microcontrollers (TFLite Micro) [13], which acts as an optimization compiler, applying transformations like operator fusion to the neural network graph. These compiler-level changes alter the expected runtime execution path, and therefore

the final energy usage. To maintain accuracy, our framework must take these TFLite graph optimizations into account in order to accurately estimate hardware energy usage.

2 Related Works

Our work builds directly on μNAS [8], the NAS framework introduced in the previous section. Given a hardware budget, such as a memory limit and a maximum allowed inference time, μNAS searches through many candidate CNN architectures and keeps only the ones that respect this budget while maximizing accuracy. Unlike most NAS frameworks, which target powerful GPUs or smartphones, μNAS is built specifically for MCUs, and it is this MCU focus that makes it the natural starting point for our work.

To judge whether a candidate architecture fits a speed budget, μNAS needs a way to estimate how fast it will run before ever deploying it on real hardware. It does this by counting Multiply-Accumulate (MAC) operations. A MAC is the basic arithmetic step behind almost every neural network layer: multiplying two numbers and adding the result to a running total. Given a layer's input and output sizes, μNAS calculates exactly how many such steps that layer requires, and summing this count across every layer in the network gives a single number, the total MAC count, that can be computed instantly without ever touching real hardware. The authors of μNAS use this MAC count as a stand-in for inference latency, reasoning that more arithmetic should generally take more time, and show that it correlates strongly with measured latency on their target MCU. We reuse this same MAC counting procedure, but instead of predicting latency, we pair the resulting counts with our own measured energy data and fit a Linear Regression model. This gives us a like-for-like baseline, built on the same proxy μNAS itself relies on, against which we can compare our proposed estimator.

A MAC count is fast to compute, but it is still only a proxy: it assumes that every multiply-accumulate operation costs the same fixed amount of time and energy, regardless of where in the network it happens or how it is executed. This assumption breaks down on real hardware, where factors such as memory access patterns, caching, and compiler optimizations can make two layers with identical MAC counts behave very differently in practice.

To capture this gap, more recent work learns hardware cost directly from data, rather than computing it from a fixed formula. BRP-NAS [11] predicts the latency of a candidate architecture using a Graph Convolutional Network (GCN), a type of deep learning model designed to operate on graph-structured data. Instead of treating the network as a flat list of operations, a GCN represents the architecture itself as a graph: each layer becomes a node, and each connection between layers becomes an edge, mirroring how data actually flows through the network during inference. The GCN then lets every node repeatedly exchange information with its directly connected neighbours, gradually building up a representation of the whole architecture that is sensitive to how its layers are arranged, not just how many of each type it has. Because this mapping from architecture to hardware cost is learned from real measurements, it can pick up on irregularities, such as caching or scheduling effects, that a fixed MAC-based formula would simply miss.

A second line of work takes a lighter-weight approach to the same problem. Energy Aware Neural Architecture Search [14] predicts energy consumption using a regression tree. Rather than representing the full network graph, this approach summarizes each candidate architecture with a handful of descriptive numbers, such as its layer count, filter dimensions, kernel sizes, and total MAC count, and feeds these as tabular input into the model. A regression tree makes its prediction by repeatedly splitting candidates on simple questions about these numbers (for example, "does the network have more than ten layers?"), narrowing down to a final energy estimate one split at a time. This makes the model cheap to train and easy to interpret, but because it relies on a small set of summary statistics rather than the network's actual topology, it cannot represent how individual layers are connected to one another.

Both directions show that learning hardware cost directly from measured data, rather than relying on a fixed formula, can substantially improve prediction accuracy over simple MAC-based estimates. However, neither closes the specific gap this work addresses: graph-based predictors like BRP-NAS target latency on mobile and server-grade GPUs rather than energy on MCUs, while the regression-tree approach above, despite targeting MCU energy, discards the network's topology in favour of summary statistics. This leaves an open question for MCU deployments, where compiler-level optimizations from engines like TFLite Micro make the network's structure, not just its size, a key factor in its energy use. Section 4 builds on the graph-based idea behind BRP-NAS, but applies it to real, measured energy consumption on an MCU.

3 Automated Energy Profiling Pipeline

This section outlines the systematic approach used to generate, measure, and assemble our hardware energy dataset. The pipeline is hardware- and task-agnostic:

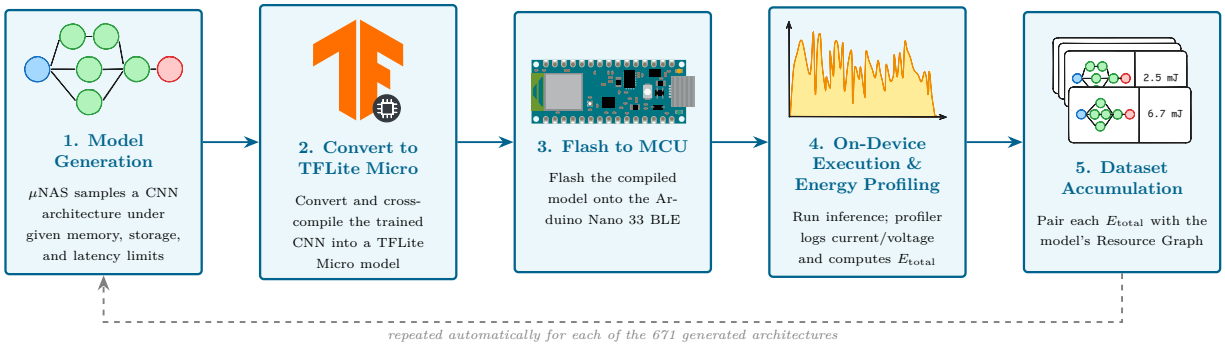

Figure 1: Overview of the automated Hardware-in-the-Loop energy profiling pipeline. Each candidate CNN generated by μNAS is automatically converted, deployed, executed, and profiled on a physical MCU, this makes up the dataset.

it can be run with different target microcontrollers and across the multiple example tasks included in the μNAS framework, such as CIFAR-10 image classification[15] and keyword spotting (KWS)[2]. The specific microcontroller, task, and power profiler used to build our dataset are reported in Section 3.6. To build a highly accurate estimator, we rely on empirical physical measurements rather than theoretical simulations. The core of our data collection is an automated Hardware-in-the-Loop (HIL) pipeline, which eliminates the manual labor required for physical hardware measurements by automating the full model lifecycle: generation, conversion, deployment, execution, profiling, and accumulation. The following subsections describe each stage of this pipeline, followed by a brief analysis of the resulting dataset. Figure 1 gives a high-level overview of the five stages and how they connect.

3.1 Model Generation

To ensure a diverse architectural search space, we utilize μNAS to automatically generate a library of distinct CNN models based on specified system requirements, including microcontroller memory, storage, and latency limitations. These models are therefore specifically optimized for the resource-constrained environment of the selected MCU. The resulting CNN Graph is saved and will later be used in the dataset.

3.2 Model Conversion

Once generated, each architecture is converted into the TensorFlow Lite Micro format and compiled directly alongside our custom profiling source code, producing a single deployable binary for the target microcontroller.

3.3 Model Deployment

The pipeline then automatically cross-compiles the resulting binary and flashes it onto the target Arduino microcontroller, making each model ready for physical measurement without any manual intervention.

3.4 On-Device Execution and Energy Profiling

After deployment, the MCU runs continuous inference using a standardized subset of the CIFAR-10 validation dataset. It is important to clarify why model accuracy is not a concern during this stage. In general, there is a well-known trade-off between accuracy and energy consumption when designing a model: smaller or more aggressively quantized architectures tend to draw less energy at the cost of predictive performance. This trade-off is already accounted for during the model generation stage, where μNAS searches over architectures under fixed resource constraints. At the profiling stage itself, however, this trade-off is irrelevant: the energy drawn by the MCU during a forward pass is determined entirely by the fixed sequence of arithmetic operations the architecture executes (convolutions, additions, pooling, etc.), not by whether the resulting predictions are correct. Since any input of the expected shape triggers the same computational path through a given architecture, the classification accuracy obtained on the validation subset has no bearing on the measured energy. The CIFAR-10 subset is therefore used only to provide consistent, representative input data that exercises the full network, not to evaluate predictive quality. During execution, a hardware power profiler measures real-time current draw and voltage across the MCU supply rails, computing the precise total energy consumption (E_{total}) per inference.

3.5 Dataset Accumulation

Finally, the system collects the individual runtime logs from each profiling run and pairs them with the model's original Resource Graph, compiling a single unified training dataset for the energy estimator.

3.6 Dataset Results

For our dataset, we ran the pipeline on the Arduino Nano 33 BLE microcontroller, profiling an image classification task on the CIFAR-10[15] dataset. Per-inference energy was measured with the Nordic Power Profiler Kit II (PPK2), which sampled the current draw and voltage across the MCU supply rails during execution. The automated profiling pipeline yielded a diverse dataset across the 671 measured neural network architectures. Figure 2 illustrates the relationship between total energy consumption per inference and the total layer count of the networks. Interestingly, there is no upward trend between layer count and energy consumption. Networks with identical layer counts exhibit highly variable energy profiles.

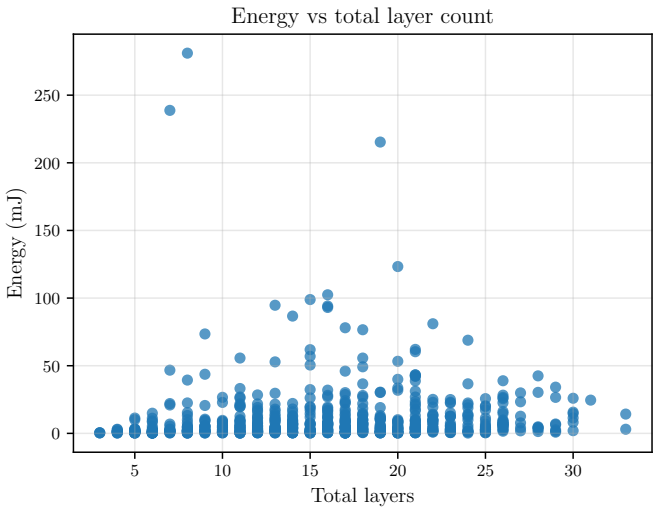

Figure 2: Energy consumption (mJ) per inference plotted against total network layer count across the generated dataset.

Figure 3 reports the total number of occurrences of each layer type, summed across all 671 profiled architectures. The histogram distinguishes only ten unique layer types, which makes the dataset look structurally narrow at first glance. This view is misleading, because a single layer type covers a wide range of computational cost depending on its configuration: a 1×1 convolution over a $32 \times 32 \times 128$ feature map performs orders of magnitude more work than the same operator over an $8 \times 8 \times 16$ map, yet both contribute one identical

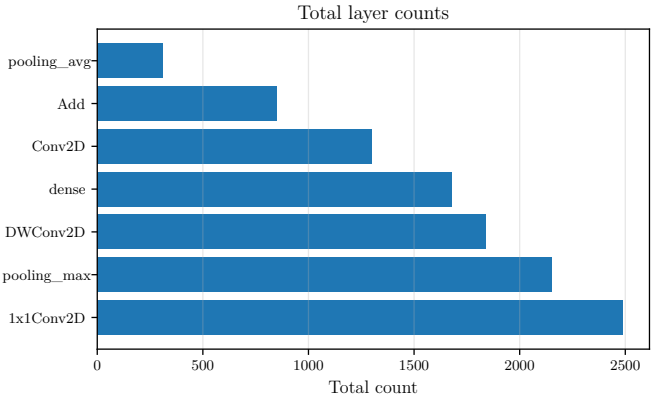

Figure 3: Collected dataset's CNN layer counts.

tick to Figure 3. Once these parameters are taken into account, the ten types expand into 1,020 distinct layer configurations spread across the 10,616 layer instances in the dataset. The substantial energy differences seen in Figure 2 therefore cannot be read off from which of the ten operators a model uses, nor from how many of each it contains. They arise from two compounding factors that a type histogram discards: the per-layer configuration (input and output shapes, channel counts, kernel size, and stride), which sets the cost of each individual operator instance, and the topological arrangement (ordering, connectivity, and repetition), which determines how those costs accumulate along the network's data flow. We therefore keep this wide range of kernel and layer parameters in mind when designing an estimator: to capture these energy differences, it must take the individual layer features into account rather than rely on layer-type counts alone.

4 Energy Estimator

An accurate hardware metric estimator is essential to evaluate searched networks. Absolute metrics like Mean Absolute Error (MAE) scale with network size, biasing the loss function toward larger architectures. To prevent high-energy architectures from dominating, we use Mean Absolute Percentage Error (Mean APE) instead. Mean APE normalizes absolute error against ground-truth values to weight all energy scales equally. Since Energy Dataset is rather small (only 671 entries) K-Folding with $K = 5$ is used to evaluate both estimators.

4.1 Baseline (MAC-only) Estimator

We first construct a baseline energy estimator based entirely on MAC operations from μNAS [8]. This baseline yields a strong goodness-of-fit with $R^2 = 0.985$, aligning

closely with the latency correlation ($R^2 = 0.975$) in the original μNAS paper.

However, global correlation coefficients conceal local predictive inaccuracies. The baseline exhibits a high Mean APE of 85.4%, with a Median APE of 24.2%. As shown in Figure 4, macro-level energy usage correlates linearly with MAC counts. Yet, this simple metric fails to capture critical hardware-level overheads. It misses memory access patterns, data movement, layer-specific execution contexts, and compilation-level TensorFlow Lite optimizations.

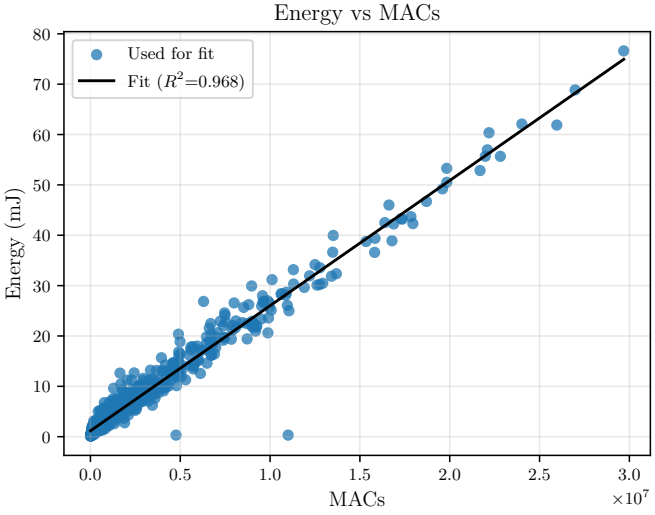

Figure 4: Energy (mJ) versus Cumulative MAC operations. MACs were estimated based on μNAS.

4.2 DAGNN Estimator

To capture the CNN topology, we introduce an estimator based on the Directed Acyclic Graph Neural Network (DAGNN) [16]. The input to the estimator is the network's Resource Graph. Edges represent data dependencies between operations. Each node represents one operation and is encoded as a 21-dimensional feature vector. This vector is the union of the configuration parameters of all CNN operator types in the μNAS search space (Conv2D, DWConv2D, Dense, Pool, and Add), so every operator fits into one fixed-length encoding. The first six dimensions are a one-hot encoding of the operator type. The remaining dimensions store the operator's configuration and output tensor shape. Large values such as height, width, and channel counts are compressed with $\log(1 + x)$. Bounded values such as kernel size, stride, and pool size are scaled to $[0, 1]$. Features that do not apply to an operator are set to zero. Table 1 lists the full encoding.

A standard GCN aggregates neighbor information and ignores edge direction. The DAGNN instead uses the direction of the DAG. It processes nodes in topological order: each node aggregates the hidden states of its predecessors, then updates its own state. This matches how tensors actually flow through the network at inference time, so the model can learn how energy costs build up along the computational path.

Table 1: Node feature vector encoding of the Resource Graph.

Dim	Feature	Applies to
0–5	Operator type (one-hot)	all
6–8	log(1+x): H, W, C/units	all
9	log(1+filters)	Conv
10–12	kernel/7, stride/4, same pad.	Conv, DWConv
13–15	batch norm, activation, bias	Conv, DWConv, Dense
16–17	log(1+units), flatten	Dense
18–19	pool size/4, is max-pool	Pool
20	equal input shapes	Add

Our final model has two DAGNN layers with a hidden dimension of 128. After message passing, the node states are pooled into one graph-level embedding. A fully connected regression head then outputs the predicted energy per inference as a single scalar. We train for 500 epochs with a learning rate of 10^{-3} and a weight decay of 10^{-4}, evaluated with Mean APE as described above. These hyperparameters come from the hyperparameter search discussed below.

A core advantage of the DAGNN is its task-agnostic flexibility. Because the model processes standardized Resource Graphs of variable size, its training is not tied to a single dataset. This allows the estimator to generalize to various network domains beyond CIFAR-10. For our deployment, we ran a hyperparameter search to optimize the DAGNN specifically for the structural distributions in the CIFAR-10 search space.

The DAGNN successfully captures the complex, nonlinear hardware interactions that a pure MAC regression ignores. As shown in Table 2, the DAGNN estimator significantly outperforms the baseline. It reduces the Mean APE from 85.4% to 16.0%.

More telling is the tail of the error distribution, which the mean and median alone conceal. The baseline's Median APE of 24.2% looks tolerable, but its 90th and 95th percentile APE climb to 237.7% and 342.6%: on its worst-predicted architectures the baseline is off by more than three times the true energy. This wide gap between the median and the tail shows that MAC counts are dependable for typical networks yet fail catastrophically on the substantial fraction of the search space. The DAGNN, by contrast, keeps its tail tight, with 90th and 95th percentile APE of only 32.3% and 37.8%, close to its 13.8% median. This robustness is what matters for NAS: a single severe energy misprediction can promote an inefficient candidate or reject an efficient one, steering the search toward suboptimal architectures.

Table 2: Performance comparison of the energy estimator models on the held-out test set. We report the Absolute Percentage Error (APE) of each model's per-inference energy predictions, summarised by the mean, median, and 90th/95th percentiles (lower is better; best values in **bold**). The MAC-only baseline degrades sharply in the tail, whereas the DAGNN stays accurate across the distribution.

Estimator Model	Mean APE	Median (50th) APE	90th APE	95th APE
Baseline (MAC-only)	85.4%	24.2%	237.7%	342.6%
DAGNN (Ours)	**16.0**%	**13.8**%	**32.3**%	**37.8**%

5 μNAS Integration

An energy estimator is only useful to a search framework if that framework can query it cheaply for every candidate it considers. This section describes how the DAGNN estimator from Section 4 is connected back into μNAS [8], so that energy becomes a search criterion alongside the memory, storage, and latency budgets that μNAS already enforces. The integration is deliberately lightweight: the estimator reuses the same Resource Graph that μNAS constructs for every candidate, and it is trained once, offline, so that consulting it during the search costs no more than a single forward pass.

5.1 A Shared Graph Interface

For every candidate architecture it evaluates, μNAS already builds a structured Resource Graph that describes the operators in the network and how data flows between them, and it uses this graph to compute the candidate's RAM usage, storage footprint, and MAC-based latency proxy. This same Resource Graph is exactly the input our DAGNN estimator consumes: as described in Section 4, each operator becomes a node encoded as the fixed-length feature vector of Table 1, and the edges carry the data dependencies between operators. Because both components already speak this representation, the estimator slots in at the point in the search loop where μNAS inspects each candidate's graph, without any additional conversion or feature-extraction step. In effect, the estimator is a function that maps a Resource Graph to a single scalar, the predicted energy per inference in millijoules, and μNAS calls it at the same place it currently consults the MAC count.

5.2 Offline Training, Online Querying

The expensive part of building the estimator happens once, ahead of any search. Collecting physical measurements through the Hardware-in-the-Loop pipeline of Section 3 and training the DAGNN on the resulting 671-model dataset is a one-time, offline cost. The trained network is then frozen and shipped with μNAS as a fixed cost model. During the search itself, obtaining a candidate's energy no longer requires deploying it to hardware; it requires only one forward pass through the frozen DAGNN, which completes in milliseconds.

This decoupling is what makes the estimator usable inside a search loop. A single NAS run evaluates a very large number of candidates, so measuring each one on the physical device would be infeasible. Moreover, a forward pass through a two-layer DAGNN is far cheaper than the work μNAS already performs to estimate a candidate's accuracy, so adding energy prediction leaves the per-candidate search cost essentially unchanged.

5.3 Energy as a Search Criterion

With predictions available for negligible cost during the search, energy can enter μNAS's decision-making in the same two ways its existing resource metrics do. First, it can act as a hard constraint: a candidate whose predicted energy per inference exceeds the device's energy budget—set by its battery capacity or its rate of energy harvesting—is discarded, exactly as μNAS already discards candidates that exceed the RAM, storage, or latency budgets. Second, it can be folded into the multi-objective fitness that μNAS optimizes, allowing the search to trade a small amount of accuracy for a meaningful reduction in energy rather than treating the budget as a hard wall.

5.4 Retargeting and Modularity

Because the estimator depends only on the standardized Resource Graph and a frozen set of weights, it is modular with respect to both the search and the target hardware. Retargeting μNAS to a different microcontroller does not touch the search code: the profiling pipeline of Section 3 is re-run on the new device to collect a fresh dataset, a new DAGNN is trained on it, and the resulting weights are swapped in.

6 Responsible Research

Our framework focuses on small-scale networks optimized for stand-alone inference on edge devices. Because the inference is performed entirely locally without sending data to the cloud, it supports localized inference in-

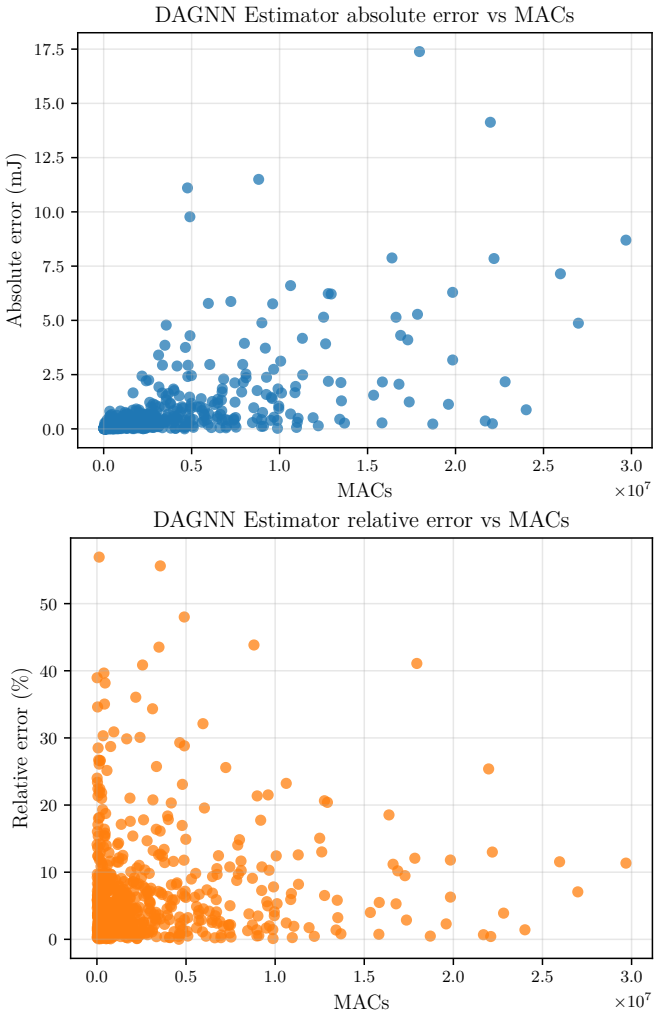

Figure 5: DAGNN Estimator's Absolute error (top) and Relative error (bottom) versus MACs. MACs were estimated based on μNAS.

stead of more expensive on-cloud inference. This localized processing creates a less pollutant (in both energy and signal emission sense) inference pipeline while simultaneously improving data privacy.
Our Energy dataset as well as, our automated pipeline are publicly available.

7 Conclusion and Future Work

This paper presented an empirical approach to estimating the energy consumption of neural networks on resource-constrained microcontrollers. We built an automated Hardware-in-the-Loop pipeline that generates, deploys, and profiles models on real hardware, and used it to collect an energy dataset of 671 CNN architectures on an Arduino Nano 33 BLE. We first evaluated a baseline estimator built on the layer-wise MAC counts from μNAS [8]. MAC counts correlate strongly with energy at the macro level ($R^2 = 0.985$), but they are imprecise for individual networks, with a Mean APE of 85.4%.

To close this gap, we introduced a DAGNN-based estimator built on Thost and Chen [16]. By operating on the network's Resource Graph, the DAGNN captures structural properties that raw MAC counts miss, such as data flow between operators and the effects of TensorFlow Lite Micro optimizations. The DAGNN reduces the Mean APE from 85.4% to 16.0%, showing that accurate, hardware-aware energy estimation is possible. Just as importantly, it collapses the baseline's heavy error tail, lowering the 95th percentile APE from 342.6% to 37.8%, so the estimator stays reliable across the search space rather than only on average.

These results make energy a practical search objective for NAS on microcontrollers. Because the estimator operates on standardized Resource Graphs, it is not tied to a single task or dataset, and the profiling pipeline can be re-run to retarget new hardware.

Future work can focus on the following three areas:

- **Cross-Hardware Testing:** Evaluating the DAGNN's transfer-learning capabilities on other microcontroller architectures.
- **Multi-Task Evaluation:** Validating the task-agnostic capabilities of the estimator across diverse workloads beyond CIFAR-10, such as Keyword Spotting (KWS).
- **Non-random Dataset:** Currently the automated pipeline generates unique, but random CNNs. A specific set of NNs might help the estimator to train more accurately or with a smaller dataset.

References

[1] Yexu Zhou, Haibin Zhao, Yiran Huang, Till Riedel, Michael Hefenbrock, and Michael Beigl. TinyHAR: A lightweight deep learning model designed for human activity recognition. In *Proceedings of the 2022 ACM International Symposium on Wearable Computers*, pages 89–93, 2022. ISWC '22 Best Paper Award.

[2] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers, 2017. arXiv:1711.07128.

[3] Juan P. Dominguez-Morales, Lourdes Duran-Lopez, Daniel Gutierrez-Galan, Antonio Rios-Navarro, Alejandro Linares-Barranco, and Angel Jimenez-Fernandez. Wildlife monitoring on the edge: A performance evaluation of embedded neural networks

on microcontrollers for animal behavior classification. *Sensors*, 21(9):2975, 2021. Special Issue: Sensors and Artificial Intelligence for Wildlife Conservation.

[4] Federica Zonzini, Francesca Romano, Antonio Carbone, Matteo Zauli, and Luca De Marchi. Enhancing vibration-based structural health monitoring via edge computing: A tiny machine learning perspective. In *Proceedings of Quantitative Nondestructive Evaluation (QNDE)*, volume 85529, page V001T07A004. American Society of Mechanical Engineers, 2021.

[5] Zohra Dakhia, Alessia Lazzaro, Mohamed Riad Sebti, Mariateresa Russo, and Massimo Merenda. On-device federated learning for energy-efficient smart irrigation. *Electronics*, 14(21):4311, 2025.

[6] Hao Liu, Qing Wang, and Marco Zuniga. Solarml: Optimizing sensing and inference for solar-powered tinyml platforms. In *2025 Design, Automation Test in Europe Conference (DATE)*, pages 1–7, 2025.

[7] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. Machine learning for microcontroller-class hardware: A review. *IEEE Sensors Journal*, 2022. arXiv:2205.14550.

[8] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. nas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, page 70–79, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Dhanusai Kanteshwara and Yogesh Choukiker. Energy aware neural architecture search, 04 2026.

[10] Hao Liu, Qing Wang, and Marco Zuniga. Instmeter: An instruction-level method to predict energy and latency of dl model inference on mcus, 2026.

[11] Łukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Brp-nas: Prediction-based nas using gcns, 2021.

[12] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Celine Lin. Hw-nas-bench:hardware-aware neural architecture search benchmark, 2025.

[13] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. TensorFlow Lite Micro: Embedded machine learning on TinyML systems, 2021. Proceedings of Machine Learning and Systems (MLSys), arXiv:2010.08678.

[14] Dhanusai Kanteshwara and Yogesh Choukiker. Energy aware neural architecture search, 04 2026.

[15] Felipe O. Giuste and Juan C. Vizcarra. Cifar-10 image classification using feature ensembles, 2020.

[16] Veronika Thost and Jie Chen. Directed acyclic graph neural networks, 2021.