Neuro-C: Neural Inference Shaped by Hardware Limits

Diletta Romano
diletta.romano@ri.se
Uppsala University and RISE
Sweden

Luca Mottola
luca.mottola@polimi.it
Politecnico di Milano,
Uppsala University, and RISE
Italy and Sweden

Thiemo Voigt
thiemo.voigt@angstrom.uu.se
Uppsala University and RISE
Sweden

Abstract

We present *Neuro-centric Networks* (Neuro-C), a neural network architecture we design to eliminate multiply-accumulate operations for efficient inference on ultra-low-power microcontrollers (MCUs). Although some MCUs include specialized hardware for neural acceleration, many ultra-low-power MCUs do not, requiring neural networks to align with limited compute and memory resources. Rather than compressing existing models or assuming dedicated hardware, Neuro-C integrates hardware constraints directly into the architecture, effectively shaping the network design around the limitations of the target platform. We shift the computational burden from connections to neurons and encode connectivity with a fixed ternary adjacency matrix, overcoming the bottleneck of matrix multiplications and large weight storage. This design enables a specialized inference kernel implementation that reduces memory usage and latency through pointer-based traversal and sparse dynamic memory allocation, complex control flows, and index decoding logic common in sparse or compressed models. Experimental results show that Neuro-C achieves accuracy *comparable to or better than* standard multilayer perceptrons across multiple datasets, while reducing inference latency and program memory usage by up to 90%. Compared to conventional ternary neural networks, Neuro-C provides improved convergence and accuracy under identical architectural settings, with negligible impact on inference latency.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Embedded software**.

Keywords: TinyML, Embedded Systems, Ultra-Low-Power MCUs,Ternary Neural Networks, Cortex-M0

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769380

ACM Reference Format:
Diletta Romano, Luca Mottola, and Thiemo Voigt. 2026. Neuro-C: Neural Inference Shaped by Hardware Limits. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3767295.3769380

1 INTRODUCTION

Ultra-low-power microcontrollers (MCUs) are widely deployed in embedded IoT systems but are characterized by severely limited memory budgets while lacking hardware features such as floating-point support [40]. Deploying neural inference directly onto such MCUs enables intelligent autonomous systems, such as battery-powered BLE nodes that detect environmental events locally [30], but requires operating within tight energy and memory budgets [27].

Problem. Despite advances in quantization and optimized kernels, deploying conventional TinyML inference, such as quantized multilayer perceptrons (MLPs) or convolutional networks (CNNs), remains challenging on ultra-low-power MCUs [10]. Multiply and accumulate (MACC) operations and dense weight matrices rapidly exhaust the limited computing and memory resources typically available on these devices. Existing TinyML frameworks, including ARM's CMSIS-NN [9] and TensorFlow Lite Micro (TFLM) [13] that provide runtime support for TinyML inference, consequently rely heavily on relatively large memory budgets and hardware acceleration, for example, hardware Digital Signal Processing (DSP) or Floating Point Units (FPUs), besides dedicated Neural Processing Units (NPUs) [49].

Common system-on-chip implementations for sensor nodes, nonetheless, must prioritize scarce die area to pack radios, power management, and advanced I/O, severely constraining memory and compute resources [7] and inherently complicating the integration of low-power NPUs. Even optimized fixed-point kernels [18] quickly exceed available memory, driving inference latency into hundreds of milliseconds and surpassing tight energy budgets. Techniques like pruning and quantization reduce memory footprint but cannot eliminate the overhead of thousands of scalar multiplications, nor can they fully mitigate branch-heavy loops and irregular memory access patterns common in generic inference implementations [20]. As a result, developers resort to hand-crafted pointer arithmetic and fixed loops, still incurring the cost of extensive matrix traversals and inefficient repeated memory loads [4].

To further complicate matters, existing TinyML frameworks often demand an underlying system-level support for scheduling and memory management, often in the form of RTOS libraries [19], adding to the memory overhead.

Contribution. We take a different stand compared to the current state of affairs. Extreme resource constraints require fundamentally different solutions tailored to bare-metal scenarios. Rather than adapting a conventional model to fit the system or relying on specialized hardware, *we tailor the model architecture to the system from the start by building hardware constraints directly into the model.*

Intuitively, we "mould" the neural network around the specific features of the MCU. We treat compute and memory features of the target platform not as limitations, but as design parameters, building upon the hardware's strengths, such as fast integer addition, low-power sequential access, and efficient loop execution, while avoiding operations that map poorly to the target MCU, such as floating-point arithmetic or large matrix multiplications. As a result, we achieve efficient inference without relying on compression techniques, pruning, or custom silicon.

We make this argument concrete with Neuro-C, a neural network architecture that eliminates costly general-purpose MACC operations. We restrict layer connectivity to a ternary adjacency matrix, that is, each neuron either does not connect, or connects with a fixed ±1 weight, and then assigns a single learned weight to the neuron's output as a scaling factor. This means that the burden of variability is shifted from individual connections to the neurons themselves.

Benefits. The structural design of Neuro-C yields a sparse, pointer-friendly computation pattern: a neuron sums contributions from a select few inputs, then multiplies once by its scale. By avoiding full matrix multiplications or convolutions, the model sidesteps the need for specialized MACC support. The execution time is entirely predictable: for a given network size, the latency is fixed, with no data-dependent variation. Memory accesses follow a simple pattern, streaming through pointer tables and input arrays, which a compiler can prefetch or optimize aggressively.

This yields much lower time and energy demands per single inference: we eliminate the MACC unit toggling and reduce memory footprint, which lowers program and data memory access energy. The use of fully connected layers, as opposed to convolutional layers, is a deliberate choice to simplify memory access patterns on MCUs.

Performance. We evaluate our design based on empirical results across multiple implementation options, using an STM32F072RB MCU (ARM Cortex-M0 core, 16KB RAM, 128KB Flash) [46]. We use a fake quantization training step via Larq [16], which produces an adjacency matrix that selects only the fundamental connections during learning. Once training is complete, models are quantized and evaluated on a regular machine over the full dataset to determine accuracy. The quantized weights are then loaded onto the target system, where we measure inference latency, commonly used also as proxy for energy consumption on ultra-low-power MCUs due to the absence of DVFS functionality [32], and memory footprint.

We specifically test Neuro-C on three datasets against conventional multi-layer perceptron networks (MLPs) and ternary neural networks (TNN), focusing on TinyML benchmark tasks with increased complexity: MNIST, FashionMNIST and CIFAR5. For both Neuro-C and standard MLPS, the model size grows almost linearly with the accuracy gain, demonstrating the necessity of a trade-off between the two. Across the different settings we test, Neuro-C brings around 90% of gain both in inference latency and memory footprint. The smallest MNIST model, with an accuracy of over 97%, runs in just 5ms and 3KB of program memory, compared to 43 ms and 31 KB of the smallest MLP that reaches the same accuracy. When pushing for over 99% accuracy, the MLP model is no longer deployable as it does not fit the available program memory, while the Neuro-C counterpart runs in 40ms and 20KB of program memory. We observe the same pattern for all other datasets.

Compared to a conventional TNN, our architecture shows improved robustness and accuracy. When the per-neuron scaling factor w_j is removed by the Neuro-C best-performing configuration, yielding a standard TNN structure, the model fails to converge on CIFAR5 and suffers accuracy drops of 2.5 and 3.5 percentage points on MNIST and FashionMNIST, respectively. This demonstrates that w_j plays a critical role in stabilizing training. Importantly, the overhead is negligible: inference latency increases by only 0.5 ms over a baseline of 50 ms, and program memory usage increases by less than 500 B over a total of approximately 20 KB.

The rest of the paper is structured as follows. Sec. 2 reviews the hardware constraints of ultra-low-power MCUs and motivates the need for system-aware model design. Sec. 3 introduces the Neuro-C architecture, detailing its per-neuron scaling, ternary connectivity, and static execution model. Sec. 4 describes the runtime implementation and deployment strategy on the target platform. Sec. 5 presents experimental results across multiple datasets, comparing Neuro-C with MLPs and TNNs. Sec. 6 discusses the limitations of this work, pointing out future developments and opportunities. Sec. 7 positions our work within the existing literature, and Sec. 8 ends the paper with brief concluding remarks.

Neuro-C source code is available at https://github.com/diletta-romano-rise/Neuro-C.

2 BACKGROUND AND MOTIVATION

Deploying neural inference on MCUs is a focal point of TinyML [31]. The appeal of deploying neural inference on resource-constrained edge devices is multifold. First, energy efficiency across the entire application operation: MCUs operating in

Class	Key features	Memory	Example
Low	8/16/32-bit core, no FPU, no DSP/SIMD	<128 KB RAM, <512 KB Flash	STMicroelectronics STM32C0/F0/L0 (Cortex-M0/M0+) [43, 45, 47]
Medium	32-bit core, single-precision FPU, basic SIMD	128–512 KB RAM, 512 KB–2 MB Flash	NXP Kinetis K series (Cortex-M4) [35]
Advanced	32-bit core, double-precision FPU, vector SIMD, optional cache	>512 KB RAM, >2 MB Flash	Renesas RA8D1 (Cortex-M85) [37]

Table 1. Qualitative analysis of MCU resources

the tens of MHz with milliwatt power usage enable battery-powered [31] or even energy-harvesting deployments [1, 3, 11]. Second, low-latency responses: running inference locally avoids the unpredictability of network latency [23], a crucial factor for sensor-based systems that require immediate reactions. Integration and cost are further concerns: many IoT platforms exist that use a SoC combining a lightweight Cortex-M core alongside wireless radios, plus tightly integrated sensors [22]. Finally, preserving privacy: by performing basic machine learning tasks locally on the MCU, the device avoids transmitting raw sensor data, which may contain sensitive information. Instead, only high-level, task-relevant outputs are sent, reducing both communication bandwidth and potential privacy leakage [23].

The challenge lies in the severe architectural and computational constraints of ultra-low-power MCUs: first and foremost, their limited memory and computing power due to the frequent lack of sophisticated hardware features. This makes regular deep learning models difficult to deploy without significant optimizations [31].

Existing platforms. MCUs vary widely in their ability to support neural network inference, due to differences in hardware features. Table 1 provides a qualitative classification.

Arithmetic support determines which operations can be performed efficiently or at all: MCUs without a floating-point unit (FPU) must emulate floating-point operations in software, resulting in significant overhead in both latency and code size. Similarly, the lack of hardware MACC instructions forces dense layers to rely on explicit loop-based implementations, which are slow and instruction-heavy. SIMD extensions, when present, improve throughput by enabling vectorized computation.

At one extreme of the spectrum are 8- or 16-bit devices, or lightweight 32-bit MCUs with no FPU or DSP support, often providing less than 64KB of RAM. These platforms must execute all neural computations entirely in software using basic integer arithmetic, which restricts deployment to shallow, heavily quantized models with limited parameter counts. For example, Cortex-M0 cores require fully software-based kernels even for small-scale tasks [21].

As memory increases into the hundreds of kilobytes and cores begin to support 32-bit floating-point arithmetic and basic SIMD, more complex neural architectures become feasible. More capable MCUs, based on cores like Cortex-M7 or ARM's Helium extensions, offer vector support and larger memory budgets, blurring the boundary between MCUs and application processors. For instance, MCUNet [29] demonstrates that, through hardware–software co-design, it is possible to run accurate image classifiers on a Cortex-M7 device with just 320KB of SRAM. However, MCUNet relies on hardware configurations at the upper end of the spectrum.

Hardware accelerators. To improve the efficiency of neural inference, modern MCUs embed forms of hardware acceleration. Starting at the Cortex-M4 core, all ARM Cortex-M cores include DSP extensions, such as MACC instructions and saturating arithmetic, that accelerate fixed-point operations directly in the MCU pipeline [8]. These instructions are heavily used by frameworks like CMSIS-NN to speed up convolutions and dense layers [9].

In addition to core-level enhancements, some MCUs embed dedicated accelerator blocks alongside the processor. The MAX78000 [6] MCU, for example, includes a fixed-function CNN engine that offloads convolution layers to a specialized compute unit, reducing inference latency and energy consumption. More flexible designs, such as Renesas's DRP-AI [37], combine a fixed MACC array with a reconfigurable controller to support a wider range of layer types and preprocessing tasks. These architectures offer substantial efficiency improvements during inference, but come with increased system integration and software development complexity, especially in the need to use separate toolchains or even different programming languages for machine learning inference as opposed to the general application logic [6].

Most importantly, the deployment of embedded accelerators is limited in embedded sensing scenarios due to inefficient application-wide energy performance [42]. Although accelerator blocks greatly improve energy efficiency during inference, other key application functionality, including I/O necessary for sensing and data transmission, suffer because of high power consumption of the cores they are attached to. As an example, the Cortex-M4 core onboard the MAX78000 consumes roughly four times the power of a Cortex-M0 core during I/O.

Existing ultra-low-power MCUs, however, lack the hardware infrastructure required to integrate and operate an accelerator. They rely on simple bus protocols, often have no cache, and no mechanisms for dynamic memory allocation or peripheral-level scheduling. Supporting even a lightweight accelerator would require additional logic for memory interfacing, clock gating, and interrupt handling, components that would exceed the area and power budget of these platforms [41].

Tools and system support. To support neural networks on MCUs, ARM provides the CMSIS-NN [9] library, which offers highly optimized kernels for common operations using fixed-point arithmetic. CMSIS-NN is tuned to each Cortex-M flavor: on a Cortex-M4/M7, it exploits the DSP instructions and SIMD parallelism to accelerate inference, whereas on a Cortex-M0, it is forced to use plain C implementations, since no special MAC/SIMD hardware is available. In practice, this means CMSIS-NN can shrink inference latency and memory footprint significantly on a Cortex-M4 core, but yields limited performance on Cortex-M0 devices.

Beyond CMSIS-NN, a number of higher-level toolkits exist to ease embedded system AI deployment. TensorFlow Lite for MCUs [17], STMicroelectronics' STM32Cube.AI [44], Edge Impuls [14], and other end-to-end pipelines take a trained model as input and generate highly-optimized, platform-specific implementations for a target MCU. They provide user-friendly workflows and often integrate with vendor SDKs, but tend to focus on capable targets, such as ARM Cortex-M4/M7 cores. As observed by Hernandez-Gonzales et al. [21], none of the mainstream solutions truly addresses extremely resource-constrained MCUs.

3 NEURO-C

We present the architecture and training methodology of Neuro-C. In doing so, we walk the reader through the process that takes the system-level constraints as the seed for the different design decisions, specifically, the need to support static memory allocation, integer-only execution, and predictable control flow on ultra-low-power MCUs.

We begin by rethinking the structure of neural computation: instead of assigning weights to individual connections, we shift this responsibility to the neurons themselves, allowing a simplified inference path based on pointer-based accumulation and per-neuron scaling. We then consider how to define the connectivity pattern, encoded as a ternary adjacency matrix, in a way that maximizes accuracy while minimizing parameter and access complexity. We compare several strategies and we motivate our choices based on both performance and implementation efficiency. Finally, we justify the use of fully connected (FC) layers over convolutional ones, not on theoretical grounds alone but by analyzing memory layout, dataflows, and runtime behavior on the target hardware. This choice strikes a trade-off between structural compactness and system-level simplicity, one that becomes essential when deploying on devices with no cache, SIMD, or MACC support.

Taken together, these design decisions form a coherent strategy for building neural systems that are not just small, but structurally aligned with ultra-low-power MCUs.

3.1 Structure

We conceive Neuro-C as a redesign of conventional neural network structures. The core idea is to shift the computational burden from the connections to the neurons themselves.

Each neuron in a given layer receives input from a sparse subset of neurons in the previous layer, as defined by an *adjacency matrix*. Instead of applying independent weights per connection, inputs are first summed and then scaled by a neuron-specific weight. This results in the following neuron activation function

$$o_j^{(l)} = f\left(w_j^{(l)} \cdot \sum_i a_{ij}^{(l)} \cdot o_i^{(l-1)} + b_j^{(l)} \right) \tag{1}$$

where

- $o_i^{(l-1)}$ is the activation from neuron i in layer $l-1$,
- $a_{ij}^{(l)} \in \{-1,0,+1\}$ defines the neuron connectivity,
- $w_j^{(l)}$ is the weight assigned to neuron j,
- $b_j^{(l)}$ is the neuron-specific bias term,
- $f(\cdot)$ is the activation function applied to the neuron.

Given the above, we can express the entire layer as

$$o = f(\mathrm{diag}(w)Ax + b) \tag{2}$$

where

- o is the output vector,
- $\mathrm{diag}(w)$ is a diagonal matrix with neuron weights w,
- A is the adjacency matrix,
- x is the input vector,
- b is the bias vector.

The key benefit of this approach lies in its ability to redistribute computational complexity while preserving the network's expressive power. Rather than relying on individual connection weights, each neuron aggregates its inputs based on a structured connectivity pattern and applies a neuron-specific scaling factor. This setup allows the network to approximate complex decision boundaries by leveraging redundancy and increased network depth. This hypothesis suggests that, with sufficient redundancy, the system can learn representations that are functionally equivalent to those of traditional weighted networks while significantly reducing computational overhead. We return to this with quantitative evidence in Sec. 5.

3.2 Adjacency Matrix

The design of the adjacency matrix plays a central role in defining the connectivity structure of the network. To retain

only the most informative connections while maintaining efficiency, we explored three different strategies.

The first strategy relies on random initialization. In its simplest form, connections are sampled independently with a fixed probability, resulting in fully unstructured sparsity. Alternatively, we can use a constrained random variant, where each neuron is assigned a fixed number of input connections, selected randomly but uniformly across the input space. While both variants introduce diversity in connectivity, their lack of structural bias can lead to suboptimal performance due to poor coverage of relevant input features or inefficient representation capacity.

A second approach is based on spatial locality. Here, connections are limited to neurons that are "nearby" in some predefined topology, typically based on index distance. This mimics the behavior of convolutional layers by enforcing local receptive fields. Such spatial structures are intuitive for image-based tasks, where local patterns dominate.

The third strategy uses quantization-aware training, where full-precision weights are maintained as latent variables during training. At each forward pass, these latent weights are quantized into ternary values (e.g., -1, 0, +1), and only the quantized version is used for inference. This allows the model to iteratively refine the connectivity pattern based on training dynamics, effectively learning which connections to retain. Compared to fixed strategies, this approach provides a better balance between flexibility and structure, allowing sparsity to emerge naturally as a function of learning rather than being fixed as a design-time decision.

To empirically evaluate these strategies, we perform experiments on a digit classification task using the *digits* dataset (8×8 grayscale images) [5]. We use a single hidden-layer architecture in all experiments, which run on standard compute infrastructure without any hardware constraints. For each strategy, we execute a grid search over various sparsity levels and hidden layer sizes. In the case of random connectivity, we include both fully probabilistic and constrained variants. The total number of parameters is defined as the sum of neurons and the non-zero entries in the adjacency matrix.

As shown in Figure 1, quantization-based connectivity yields the best performance for a given parameter count. It achieves higher accuracy with fewer neurons, indicating more efficient use of representational capacity.

Based on these findings, we employ the quantization-based strategy hereafter. It accurately captures the data features while determining connectivity and remaining compatible with our system-level constraints.

3.3 Fully Connected Structures over Convolutional Layers

On many embedded platforms, CNNs are preferred due to their parameter sharing and spatial efficiency. However, on ultra-low-power MCUs such as the Cortex-M0, this rationale

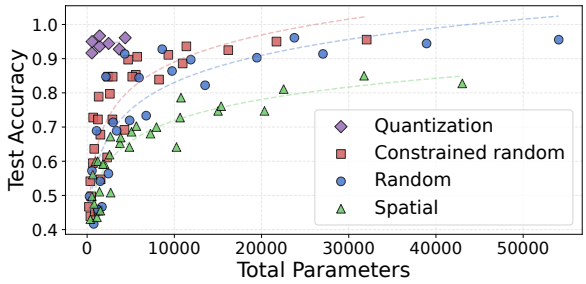


Figure 1. Test accuracy against the total number of parameters for different adjacency matrix strategies on the *digits* dataset. *The quantization-based connectivity achieves the highest accuracy given the number of parameters, demonstrating its superior expressivity under strict sparsity constraints.*

does not hold. These devices lack DSP extensions, SIMD support, and sufficient RAM to support optimized convolution routines or intermediate buffers. As a result, standard CNN implementations fall back on software-based `im2col` transformations, which inflate both memory usage and inference latency.

In contrast, we can implement FC layers with direct pointer traversal and static memory layouts, without reshaping input tensors or managing memory strides. This simplicity translates into predictable inference latency and lower memory overhead.

A standard convolution applies a kernel of size $S \times S$ over an input of spatial size $N \times N$ with C input channels, producing an output of size $M \times M$, where

$$M = N - S + 1 \tag{3}$$

Since convolutions involve local receptive fields, on lightweight hardware, the input must be transformed into a matrix using the `im2col` operation. This transformation converts the input tensor into a *flattened matrix* of size

$$(C \cdot S^2) \times M^2 \tag{4}$$

where each row is a flattened receptive field of the convolution, and each column represents an output spatial location.

We reshape the corresponding kernel of size $C \times S \times S \times K$, where K is the number of filters, into a weight matrix of size

$$K \times (C \cdot S^2) \tag{5}$$

The convolution operation is then computed as a matrix-matrix multiplication (GEMM)

$$(K \times (C \cdot S^2)) \times ((C \cdot S^2) \times M^2) = K \times M^2 \tag{6}$$

Thus, the number of MACC operations required per convolutional layer is

$$\text{MACCs}_{\text{CNN}} = K \cdot C \cdot S^2 \cdot M^2 \tag{7}$$

Additionally, this requires storing the `im2col` matrix, whose size depends on the output spatial resolution based on 4. This

intermediate storage introduces significant memory overhead, particularly when C or S is large. For an FC layer with N_{in} input neurons and N_{out} output neurons, each output is computed as a weighted sum of all input neurons. The number of MACC operations is

$$\text{MACCs}_{\text{FC}} = N_{\text{out}} \times N_{\text{in}} \tag{8}$$

The ratio of MACC operations between a convolutional and an FC layer is

$$\frac{\text{MACCs}_{\text{CNN}}}{\text{MACCs}_{\text{FC}}} = \frac{K \cdot C \cdot S^2 \cdot M^2}{N_{\text{in}} \cdot N_{\text{out}}} \tag{9}$$

For many CNN architectures, especially in early layers, the feature map's spatial dimensions remain close to the input size, thus $M \approx N$. If we consider this approximation and the same input size for the two layers, meaning $M^2 \approx N^2 = N_{in}$, we can write

$$\frac{\text{MACCs}_{\text{CNN}}}{\text{MACCs}_{\text{FC}}} = \frac{K \cdot C \cdot S^2 \cdot N_{in}}{N_{in} \cdot N_{\text{out}}} = \frac{K \cdot C \cdot S^2}{N_{\text{out}}} \tag{10}$$

Beyond their computational cost, FC layers present notable advantages in terms of implementation simplicity. They enable sequential memory access, as inputs can be processed directly without requiring complex memory layouts or stride computations. Unlike convolutional layers, FC layers do not require padding management and `im2col` transformations, which are often needed to convert convolutions into efficient matrix multiplications. Additionally, the direct and regular dataflow of FC layers facilitates their implementation on MCUs, where predictable memory access patterns are crucial for energy-efficient execution.

To evaluate the efficiency of FC structures against their convolutional counterpart, we set up an experiment comparing the inference latency of these two layer types under equal MACC conditions, according to Equation Eq. 10. For the same input size of $16 \times 16 = 256$, we measure the inference latency when N_{out} of the FC layer matches $K \cdot S^2$ of the CNN layer, assuming input channels $C = 1$. This approach allows us to isolate and observe the effects of implementation choices independently of MACC count.

Fig. 2 shows the results on a Cortex-M0 MCU for two specific cases. CNN1 (CNN2) matches the number of MACC operations for FC1 (FC2). The plot shows how FC layers exhibit lower inference latency compared to their convolutional counterparts, regardless of the dimension. This is due to simpler memory accesses and control flows, which gracefully map to the feature of the MCU at hand, which lacks SIMD or convolution acceleration.

Note that FC layers are, in principle, more expressive than convolutional layers given the same number of MACC operations. Therefore, any performance gain we can reap for CNNs should be considered alongside the potential accuracy and representational advantages offered by MLPs.

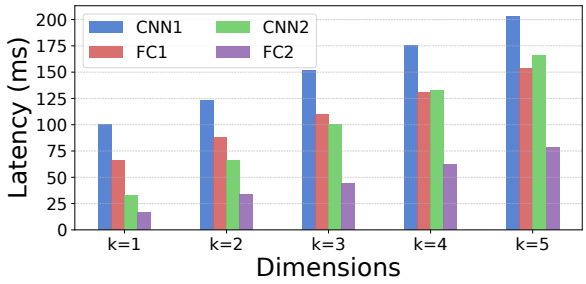


Figure 2. Inference latency of convolutional and FC layers on a Cortex-M0 MCU. *The case of CNN1 (CNN2) matches FC1 (FC2) in the number of MACC operations. FC layers consistently achieve lower latency due to simpler memory access and control flow, making them more suitable for ultra-low-power MCUs without SIMD or convolution acceleration.*

Given these empirical results, we opt for a fully connected structure for Neuro-C. This choice is driven by simpler memory access patterns, reduced computational overhead in ultra-low-power MCUs, and its flexibility in representing complex functions despite the observed efficiency trade-offs.

3.4 Comparison with TNNs

Although the design of Neuro-C shares similarities with conventional TNNs, the way it is obtained follows a fundamentally different reasoning. Standard techniques for building TNNs are derived from binary networks: rather than applying a simple sign function to the weights, ternary quantization introduces thresholds and optimizes them to minimize the quantization error, often in a post-training step. When training from scratch, the same concerns of BNNs apply to TNNs: convergence is usually facilitated by retaining the floating-point representation for some layers, and stability strongly depends on batch normalization. However, batch normalization cannot be folded into ternary weights and must be retained in inference, which makes these models unsuitable for ultra-low-power MCUs.

In Neuro-C, by contrast, stability is not enforced through auxiliary high-precision operations or sophisticated quantization procedures, but is embedded directly within the model's design. The per-neuron scaling factor acts as a built-in normalizer that enables convergence without relying on batch statistics or floating-point layers. This simple mechanism matches the core philosophy of Neuro-C: rather than adapting an existing architecture to ternary constraints through complex quantization pipelines, we shape the architecture around the limitations of Cortex-M0 devices, yielding a structure that is both efficient and deployable.

We further explore the impact of this scaling factor in Section 5.2, where we experimentally remove it and show that a pure ternary network without w_j leads to sub-optimal accuracy and, in some cases, lack of convergence.

4 CASE IN POINT: ARM CORTEX-M0

Shaping the neural network architecture to match the features of ultra-low-power MCUs requires rethinking their design not only in terms of model compression but also deep down into their concrete implementation.

As much as existing tools and system support, described in Sec. 2, apply platform-specific optimizations and techniques, we present next a platform-specific implementation of Neuro-C for the widespread Cortex-M0 MCU. We focus on this MCU as it generally represents an efficient trade-off between compute power and energy consumption [25] and is often used in SoC designs deployed in sophisticated embedded sensing applications, including energy-harvesting settings [50]. Our implementation choices are geared towards optimizing inference latency, memory footprint, and implementation simplicity, where lower latency also translates into reduced energy consumption [2, 32].

4.1 Embracing Platform Constraints

Deploying neural network inference on MCUs akin to ARM Cortex-M0 MCUs requires more than reducing model size. The architecture imposes structural limits that directly shape how inference must be implemented.

Challenges. The core lacks a floating-point unit and forces data processing to rely on fixed-point integer arithmetic. It provides no hardware support for fused MACC operations, meaning that every dot product must be implemented using explicit loops with separate multiplication and addition steps. This adds compute overhead and thus energy consumption, especially when scaled across layers.

The memory architecture imposes additional restrictions. Instructions and data share a single 32-bit AHB-Lite system bus with no caching or prefetch mechanisms, so every access to program memory or SRAM bears a direct performance cost. Instruction fetches from program memory are blocking and may require wait states depending on the state of the instruction pipeline, particularly at higher clock frequencies. RAM is severely limited, and yet debugging memory issues is extremely difficult to limited visibility [51]. This makes dynamic memory allocation extremely costly and pushes developers to statically allocate all data structures, including weights and network topology, and to access those using fixed compute structures. Intermediate representations that require buffer reshaping or runtime index computation, such as `im2col` for convolutions, quickly become impractical unless they are aggressively pruned and flattened a priori.

Control flow during inference must also be tightly controlled. The Cortex-M0 has no branch prediction, and every branch taken flushes the shallow three-stage pipeline, introducing a fixed multi-cycle penalty. Conditional statements within the main compute loop, particularly if depending on input data, introduce variability and increase latency. Recursion and deeply nested loops are discouraged due to stack limitations. Inference routines must thus be constructed through static control flow, with fixed loop bounds and no data-dependent branching. The preferred execution model is thus a sequence of shallow, possibly unrolled loops over contiguous memory segments.

Neural inference, as we argue in Sec. 2, does not live alone. Most often, the application logic combines that with sensing, energy management, and data transmission [38]. In this setting, interrupt behavior, for example, coming from sensors, further constrains the design. When an interrupt occurs, the core performs a full context save onto the main stack, and available memory must be sufficient to preserve inference state during preemption. If inference time is not tightly bounded, the system must be designed to tolerate interrupts or defer them predictably. Since the Cortex-M0 lacks hardware performance counters such as a cycle counter, fine-grained and non-intrusive profiling of inference execution is not possible. This limitation reinforces the need for statically analyzable and predictable execution paths.

Under these conditions, traditional neural network architectures must be restructured to match the execution model of the hardware. The network must be expressed as a fixed sequence of integer additions, ideally using sparse connectivity and static memory traversal. Dynamic control flows, runtime memory allocation, and index-based indirection to access memory locations must be avoided. We keep maintaining that models must be designed not only for minimal size but also for tight coupling with the instruction set and memory read/write patterns of the target device.

Key insight. Based on this discussion, the data structure used to encode sparse connectivity becomes a *first-order concern*. On a processor without indexed addressing or speculative execution, the memory access pattern and control path during inference are indeed determined directly by how connectivity is represented. A format that requires pointer dereferencing, index decoding, or dynamic traversal introduces conditional branches, data-dependent loop bounds, and additional memory loads, all of which are costly on the Cortex-M0.

In contrast, a compact encoding that enables linear iteration using fixed offsets can be mapped directly to a sequence of `load-add` instructions with no intermediate decoding. This reduces both control overhead and instruction count, and ensures that inference time remains predictable across inputs. As a result, the choice of encoding format is not simply a matter of storage efficiency, as it fundamentally determines whether inference can execute within the timing and energy constraints of the system. In the next section, we evaluate several design options in this regard and analyze their impact on both memory usage and inference latency.

4.2 Efficient Sparse Matrix Encoding

Sparse connectivity is a natural fit for low-power inference, as it reduces the number of operations and the amount of data

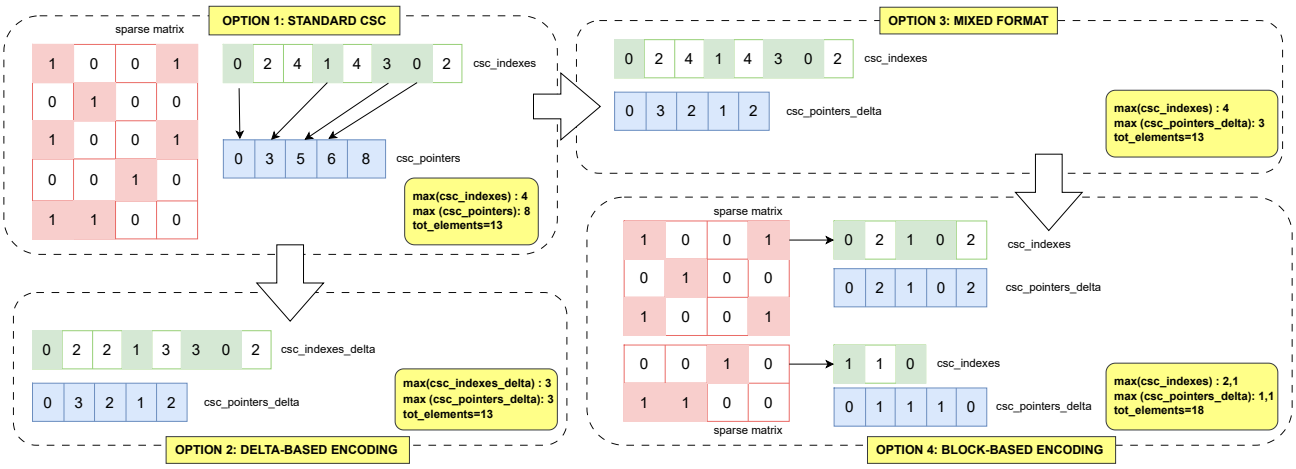


Figure 3. Encoding strategies applied to the same sparse matrix. *Each format is represented by its pointer and index arrays, showing index ranges, total number of parameters, and potential compression ratios.*

stored. However, encoding sparsity efficiently is non-trivial on devices without SIMD, FPU, or MACC support. General-purpose sparse matrix formats like Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) are designed for platforms that reconstruct full matrix rows or columns at runtime and perform indexed multiplications. These approaches introduce memory indirection and decoding steps that are costly on MCUs like the Cortex-M0.

We explore four encoding schemes for the connectivity matrix in Neuro-C. These encodings must minimize memory footprint while allowing inference to proceed without matrix reconstruction or complex runtime control flow. Each encoding stores, for every output neuron, the indices of non-zero input neurons with either positive or negative connections, separated in two disjoint index sets. Each format defines how these indices and their positions are stored and traversed.

Figure 3 shows an illustrative comparison of the four encoding schemes using a toy sparse matrix, and serves as a yardstick through the rest of this section. It highlights the structure of the pointer and index arrays in each case, showing how compression is achieved through delta offsets, shared block-local pointers, or compacted metadata. These encoding strategies form the foundation of our inference backend. We experimentally evaluate their performance in memory usage and inference latency next.

CSC baseline. The baseline scheme, shown at the top left in Fig. 3, adopts a standard CSC representation, which stores two arrays for each polarity: one containing absolute input indices, and one defining the boundaries of each column via pointers. The pointer array indicates the range of entries in the index array associated with each output neuron. This format is straightforward to implement, supports constant-time sequential traversal, and requires no decoding logic at runtime.

However, its scalability is limited by the size of the neuron index space. When either the number of input or output neurons exceeds the range of 8-bit representations, those are no longer sufficient to encode pointers and indices, and 16-bit integers must be used. As a result, the memory required quickly becomes a limiting factor in MCUs with limited memory available to store weights and structures.

Delta-based encoding. To reduce memory usage, we implement a delta-based variant, shown at the bottom left of Fig. 3. Unlike standard CSC, this format does not store absolute indices for all non-zero connections. Instead, each output neuron stores the position of its first input neuron explicitly, and all subsequent connections are encoded as relative offsets from the previous index. Column pointers do not indicate absolute positions in the index array but store the number of non-zero entries in each column.

The pseudocode in Fig. 4 shows the traversal mechanism of this structure, which becomes possible using simple pointer arithmetic. A pointer is initialized at the first index and subsequent addresses are computed by incrementing the pointer by each stored offset. This eliminates explicit decoding or index reconstruction. Each column is processed as a sequence of relative memory accesses, accumulating the values of input neurons based on the offset list.

The effectiveness of this encoding depends on the distribution of active connections. When deltas are small, indices can be stored in 8 bits, reducing memory usage and enabling faster access. However, the format does not guarantee that all offsets fall within the the 8-bit range. In layers with sparse or irregular connectivity, 16-bit deltas may still be required to avoid overflow or truncation.

Mixed formats. We also consider a compromise between simplicity and compression, shown at the top right of Fig. 3. As in the delta variant, the column pointer array stores the

```
FORWARD_DELTA(layer, INPUT, OUTPUT):

  MOV P_PTR, layer.csc_indexes_delta
  MOV ROW_PTR, layer.csc_pointers_delta

  FOR COL = 0 TO layer.output_size - 1 DO
    MOV SUM, 0
    MOV DELTA, ROW_PTR[COL + 1] #num of elements in column

    IF DELTA > 0 THEN
      MOV I_PTR, INPUT + [P_PTR] #first index is absolute
      ADD SUM, [I_PTR] #accumulate first input value

      WHILE --DELTA > 0 DO
        #follow relative offset and accumulate
        ADD SUM, [I_PTR = I_PTR + [++P_PTR]]
      END

      ++P_PTR #advance to next column
    END
    #write scaled sum to output
    MOV OUTPUT[COL], SUM * layer.scale[COL]
  END
```

Figure 4. Delta-based traversal for a column: the first index is absolute, following entries are relative offsets.

number of non-zero elements per output, but the index array retains absolute indices. This allows direct reading of each input position while still reducing the overhead associated with full-range pointers.

Compared to the baseline, this format requires less memory, and compared to the delta encoding, it avoids sequential dependencies by storing absolute indices, allowing for stateless and direct traversal.

Block-based encoding. In a further scheme, shown at the bottom right of Fig. 3, we partition the input space into fixed-size blocks. Each block maintains an independent encoding of positive and negative connections, including separate pointer and index arrays. This strategy reduces the addressable range within each block, enabling further compression and simplifying pointer arithmetic. At runtime, inference proceeds in multiple passes, one for each block, with separate index traversal and accumulation.

This layout is particularly effective when the input dimension is large and sparse connections tend to cluster within localized regions. Moreover, it is the only encoding that guarantees, by construction, that all indices remain within a fixed range. By limiting the block size to 256 inputs or fewer, we can store all indices as 8-bit integers without the risk of overflow.

4.3 Performance Trade-offs

We evaluate the four encoding formats on a Cortex-M0 MCU. We write the code in C and compile it with no operating system support. We statically allocate all memory structures and store them on flash memory. Inputs and outputs use 16-bit integers or 8-bit integers when possible; accumulations and scaling use 32-bit intermediate buffers to avoid overflow.

We implement a single-layer feedforward kernel with a fixed input dimension and sparsity ratio. We vary the number of output neurons in powers of two from 32 to 256. For each configuration, we measure the average inference time over 100 runs using TIM2, a 32-bit hardware timer configured without prescaling and clocked at the system frequency, and keep track of flash memory use.

Figure 5a shows the inference latency for each of the four implementation options. The delta-based encoding achieves the lowest latency across all output sizes. At $N_{\text{out}} = 256$, for example, it completes inference in 26 ms, compared to 32 ms for standard CSC. Mixed and block-based encoding reach 28 ms and 30 ms, respectively.

Figure 5b reports the flash memory occupation. The block-based encoding exhibits the lowest memory requirement, consuming 11.6 KB at $N_{\text{out}} = 256$, compared to 20.1 KB for standard CSC. Delta and mixed format achieve intermediate savings, but still possibly require 16-bit storage for some connections due to large indices or offsets.

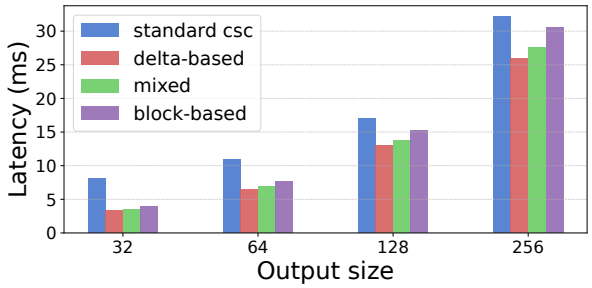
These results reflect the trade-offs between encoding complexity, flexibility, and performance. While delta-based traversal minimizes inference latency, only the block-based encoding guarantees the systematic use of 8-bit indices, making it the most memory-efficient option under all configurations. Based on these findings, we employ the block-based strategy hereafter.

5 EVALUATION

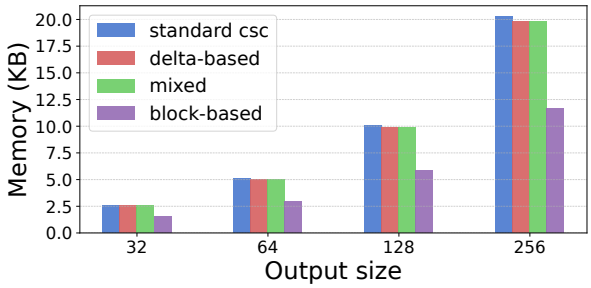
We report on an experimental evaluation of Neuro-C on a Cortex M0 core. The evidence we collect indicates that

1. Compared with MLPs models providing comparable accuracy and within the limits of available program memory, Neuro-C provides a $\approx 89\%$ speedup in inference latency, and hence in energy consumption.
2. Neuro-C achieves 99%+ accuracy in inference in cases where MLP models are not even deployable, as they exceed the available program memory.
3. When providing comparable accuracy, Neuro-C reduces the program memory occupation by one order of magnitude compared to its MLP counterpart, leaving room for other application functionality.
4. Compared to conventional TNNs configured by removing the per-neuron scaling factor w_j from Neuro-C architectures, Neuro-C consistently converges and achieves better accuracy, especially as input complexity increases, with minimal impact on latency and memory.

In the following, Sec. 5.1 describes the experimental setting, performance metrics, and baselines we compare with. Sec. 5.2 illustrates the results and key take-aways.



(a) Inference latency (ms) as a function of output size N_{out} for each encoding format. *Delta-based encoding is consistently the best performing implementation choice.*



(b) Flash memory usage (KB) as a function of output size N_{out} for each encoding format. *The block-based encoding is the most compact storage layout.*

Figure 5. Comparison of latency and memory usage for different encoding schemes.

5.1 Setting

We use an STM32F072RB[46] board running at 8 MHz, equipped with 128 KB of program memory and 16 KB of RAM. We implement all neural models in plain C, using statically allocated memory and fixed-point integer arithmetic. We compile the code using `arm-none-eabi-gcc` with `-Os` setting. The resulting implementations run on bare metal.

We measure and report three key metrics: *classification accuracy*, which we measure offline after training and `int8` quantization; *inference latency*, which we measure using the TIM2 timer on the target device; and *program memory usage*, as indicated by the size of the statically linked binary sections containing weights and inference code. In the absence of DVFS functionality [32], as is the case for most low-power MCUs, inference latency is directly proportional to energy consumption and indeed, often used as a proxy for the latter metric in low-power embedded systems [48].

We compare Neuro-C against two baselines: conventional MLPs and TNNs. The former represents widely used dense models in TinyML, while the latter shares the same architecture as Neuro-C but without the per-neuron scaling factor. This allows us to isolate and evaluate the contribution of this architectural component. Our goal is to show that Neuro-C achieves improved accuracy, lower inference latency, or both, depending on the baseline and task. We focus our comparison on these architectures as they represent the current deployment reality for Cortex-M0 platforms, where more advanced methods requiring hardware acceleration features are not practically deployable.

We train each model using the Larq framework [16], which supports quantization-aware training with binary and ternary weights. We employ manual model selection across various configurations to gain detailed insights into Neuro-C's architectural behavior. While automated search methods might be applied, our controlled manual exploration allows us to track how performance and sparsity patterns evolve across different network sizes and connectivity levels, providing observability into the architecture's dynamics beyond what automated methods would reveal. After training, we export the models to a custom inference engine, which mirrors the final network structure and supports the target deployment constraints.

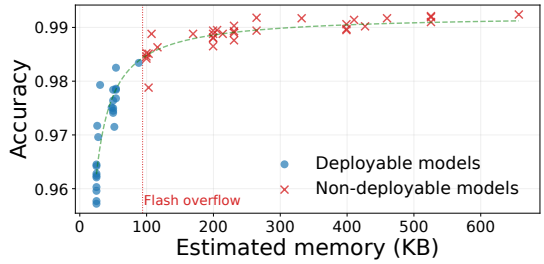
We focus on image classification datasets to ensure fair, unbiased comparison following established TinyML benchmarking practices. Unlike temporal and sensor tasks that require domain-specific preprocessing, for example, spectrogram generation for keyword spotting or feature engineering for anomaly detection, image datasets provide standardized evaluation without confounding factors that could bias results toward specific preprocessing choices [33]. In particular, we evaluate all models on three stable image classification datasets [28]: MNIST, Fashion-MNIST, and CIFAR5, the latter being a subset of CIFAR-10 restricted to the first five classes, as standard MLPs fail to achieve meaningful accuracy on CIFAR-10 [52].

5.2 Results

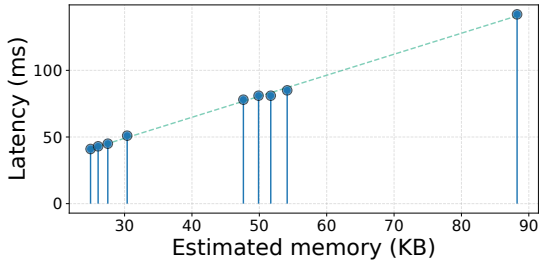
Fig. 6a shows the results of this process for MNIST. As expected, increasing the number of parameters generally leads to improved accuracy. The vertical red line separates deployable models, that is, configuration that fit within the memory constraints of our the platform, from non-deployable ones due to program memory limitations. Note that this evaluation does not consider application functionality other than the neural models themselves; practical deployments may include other functionality, such as sensor drivers and network stacks, that further limit available memory.

For the MLP configuration that may be practically deployed, Figure 6b illustrates the inference latency to assess how it scales with model size. This figure increases linearly with the number of parameters, consistently with the expectations for dense models running on constrained hardware.

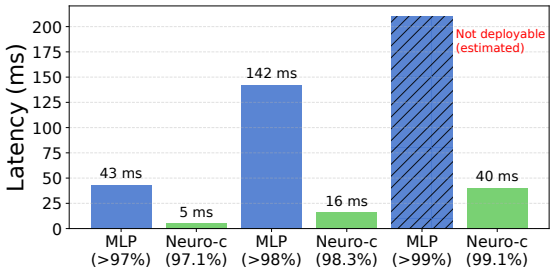
MLP comparison. First, we seek to establish a robust performance baseline in terms of validation accuracy across a range of model complexities. We perform an extensive random search over more than 50 MLP configurations by varying the numbers of layers, dropout rates, and whether batch normalization is employed.



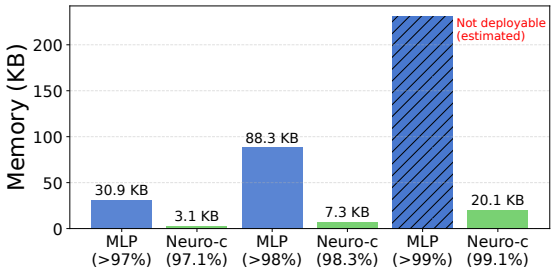
(a) Validation accuracy of different MLP models as a function of their sizes. *Models marked as non-deployable exceed the Cortex-M0 program memory. Accuracy increases with the number of parameters and thus model size.*



(b) Inference latency of deployable MLP models on Cortex-M0 as a function of their size. *Each data point represents a deployable model configuration, inference latency increases linearly with the number of parameters, and thus model size.*



(c) Inference latency of configurations with comparable accuracy. *Neuro-C models exhibit substantially lower inference latency than corresponding standard MLPs. The MLP at >99% accuracy exceeds program memory limits.*



(d) Program memory usage of configurations with comparable accuracy. *Neuro-C models require significantly less program memory compared to standard MLPs. The latter at >99% accuracy exceeds available program memory.*

Figure 6. Comparison of conventional MLP models and the proposed Neuro-C architecture on the MNIST dataset. *Neuro-C reduces inference latency and program memory consumption when providing comparable accuracy.*

To compare the performance of Neuro-C against conventional MLP models, we manually select Neuro-C models on MNIST to represent three scales: small, medium, and large. As with MLPs, we observe a trade-off between model size and accuracy. To enable a fair comparison, for each Neuro-C model, we select the smallest MLP configuration output by the random search that achieves approximately the same accuracy. For instance, the small Neuro-C model reaches 97% accuracy; we pair it with the smallest MLP configuration that exceeds 97%. This strategy allows for a fair comparison of inference latency and program memory utilization.

Fig. 6c presents a direct comparison between MLPs and Neuro-C models at comparable accuracy levels. The difference is substantial across all configurations. For models reaching over 97% accuracy, the standard MLP requires 43 milliseconds per inference, while the corresponding Neuro-C model, achieving 97.1% accuracy, completes inference in just 5 milliseconds, resulting in an 88% reduction in latency. At 98% accuracy level, the MLP model takes 142 milliseconds, whereas the Neuro-C model reaches 98.3% accuracy in only 16 milliseconds, yielding a 89% speedup. In the high-accuracy configuration, the MLP achieves over 99% accuracy but is no longer practical due to its program memory footprint, while the Neuro-C configuration, achieving 99.1% accuracy, remains deployable and runs in 40 milliseconds. These results illustrate that Neuro-C remains efficient even at the upper end of the accuracy range, where standard MLPs become impractical.

Similar observations apply to the trends in Fig. 6d investigating program memory usage. At the lower accuracy level, the MLP model requires 30.9 KB of program memory, while Neuro-C uses only 3.1 KB, representing a 90% reduction. In the 98% accuracy case, the MLP model uses 88.3 KB compared to just 7.3 KB for the Neuro-C model, again corresponding to more than 91% savings. For the highest-accuracy target, the MLP model exceeds 200 KB and cannot be deployed on the target device, while Neuro-C remains within the platform limitations at 20.1 KB. We hereby demonstrate that Neuro-C models consistently match or exceed the accuracy of MLPs while offering significantly lower inference latency and program memory usage, confirming how embracing the platform limitations within the model architecture is ultimately beneficial.

For the other two datasets, Fashion MNIST and CIFAR5, we select the best-performing deployable models for both MLP and Neuro-C. In the case of standard MLPs, this corresponds to the best model obtained through our random search procedure that still satisfies deployment constraints, particularly program memory. For Neuro-C, we manually select the most

accurate configurations we could find during a manual search process.

Fig. 7a reports the corresponding classification accuracy. Neuro-C consistently achieves higher accuracy across all three datasets. While the absolute differences may appear small, these are datasets where such margins are arguably meaningful. Note that these are not the overall best-performing MLPs in terms of accuracy, but the best-performing ones that remain deployable. Higher-accuracy MLPs exist, but they exceed the program memory of the target platform.

Despite the fact that for Neuro-C we select the largest models, that is, those with highest accuracy, the performance gap in terms of inference latency and memory is substantial. As shown in Figure 7c, the latency of Neuro-C models is consistently lower. For example, on MNIST, it drops from 140 ms of the MLP configuration to 43 ms for Neuro-C; on Fashion MNIST, it drops from around 120 ms to 30 ms; and on CIFAR5, it drops from over 100 ms to less than 50 ms. We observe similar patterns in program memory usage, reported in Fig. 7c: Neuro-C models use approximately 20-35 KB of program memory, while their MLP counterparts require 80-90 KB.

These results confirm that, even in high-accuracy regimes where both models are pushed to their deployability limits, Neuro-C maintains a clear advantage in both inference latency and program memory footprint.

TNN comparison. We compare Neuro-C with a standard ternary neural network (TNN) baseline. This comparison doubles as a means to assess the impact of the per-neuron scaling factor w_j. We obtain the TNN version by removing the scaling factor from Neuro-C, resulting in a structure where each connection has a ternary weight $a_{ij} \in \{-1,0,+1\}$, and no form of normalization or per-neuron scaling is applied.

Both Neuro-C and the TNN baseline are executed on our custom optimized inference kernel, ensuring that differences are due solely to the architectural design. The remaining architecture and training protocol are kept identical.

This comparison is motivated by the structural similarity between Neuro-C without per-neuron scaling factor and classical TNNs, which are known to face convergence issues when trained from scratch in deeper architectures. Our evaluation aims to assess whether Neuro-C addresses these limitations through the inclusion of w_j, and to what extent this impacts both accuracy and inference efficiency.

To ensure a fair comparison, we evaluate the TNN baseline using the largest and best-performing Neuro-C architecture for each dataset. This setup ensures that any performance differences can be attributed to the absence of the per-neuron scaling factor, rather than to differences in model capacity.

Fig. 8a reports the resulting classification accuracy across three datasets. For MNIST and Fashion MNIST, the accuracy performance of the Neuro-C configuration without per-neuron scaling degrades in both cases, with a drop of 2.53 and 3.55 percentage points, respectively. More critically, this configuration fails to converge entirely on CIFAR5, providing evidence of the role of w_j in stabilizing the training dynamics as the input complexity increases.

We further evaluate whether eliminating the per-neuron scaling factor leads to any meaningful improvements in latency and memory. We benchmark the same inference code of the best-performing Neuro-C configuration, with and without the scaling factor. Fig. 8b shows that removing the per-neuron scaling factor reduces latency by less than one millisecond across all datasets. Given baseline inference times of 40–50 ms, these gains are negligible.

The same holds for memory usage. In Fig. 8c, we show the difference in program memory between the two configurations. The observed reductions are 282 B on MNIST, 410 B on FashionMNIST, and 297 B on CIFAR5, relative to baseline memory footprints of approximately 20 KB.

This analysis demonstrates that the inclusion of per-neuron scaling in Neuro-C is essential for enabling convergence without batch normalization and that its removal results in both inferior accuracy and negligible inference latency savings. The w_j term is thus a critical architectural component that ensures stability and expressivity while remaining compatible with the constraints of the target architecture.

6 DISCUSSION

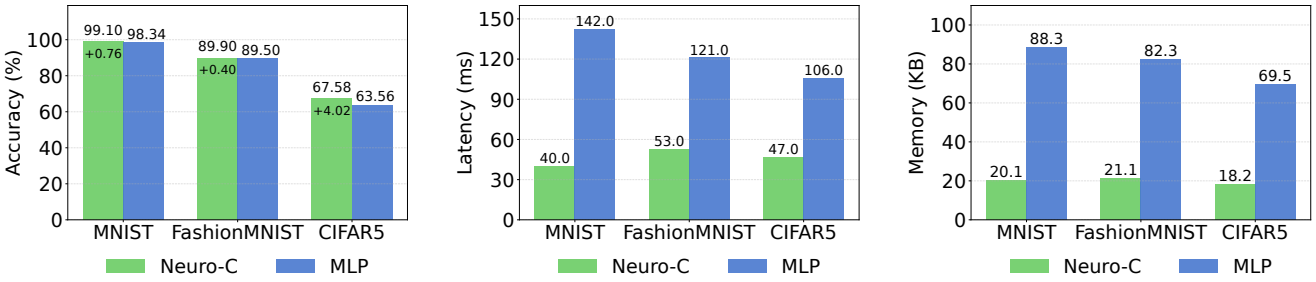
Our results demonstrate that embracing hardware constraints as first-order design principles enables efficient neural inference on ultra-low-power MCUs.

We specifically focus on image classification tasks because they provide standardized benchmarks and fair comparisons across models. Still, the co-design principles behind Neuro-C are not limited to vision. In domains such as keyword spotting or anomaly detection, where inputs often lack spatial structure, the gains may be even more pronounced.

Another defining feature of Neuro-C is its strong coupling with the hardware platform. In this paper, performance gains arise from aligning the model with the specific constraints of the Cortex-M0. On devices with different architectural features, such as cache, DSP, or SIMD units, the same design philosophy would lead to different architectural choices. This adaptability is consistent with our vision: the network should be shaped by the hardware, not vice versa.

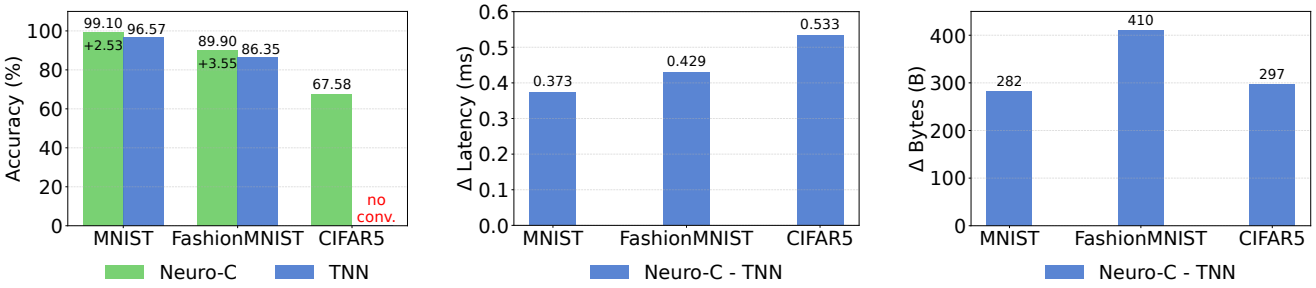
Finally, while we deliberately relied on manual model selection to gain insights into architectural dynamics, more systematic exploration could reveal additional trade-offs. Likewise, a full ablation of Neuro-C's design parameters, such as connectivity patterns, sparsity levels, or per-neuron scaling, would provide a finer-grained understanding of how each choice contributes to efficiency and accuracy.

Still, Neuro-C shows that even at the lowest end of the MCU spectrum, efficient inference is possible when models are built around hardware constraints.



(a) Accuracy. **(b)** Inference latency. **(c)** Program memory usage.

Figure 7. Performance of of the best deployable models of MLP and Neuro-C on MNIST, Fashion-MNIST, and CIFAR5. *Neuro-C achieves better accuracy, inference latency and program memory footprint, enabling efficient inference under tight resource constraints.*



(a) Accuracy. **(b)** Inference latency overhead. **(c)** Program memory usage overhead.

Figure 8. Comparison between Neuro-C and the TNN variant in terms of accuracy, inference latency increase, and program memory overhead across MNIST, FashionMNIST, and CIFAR5. *Neuro-C achieves higher accuracy, while the additional latency and memory overhead remain negligible.*

7 RELATED WORK

In light of the experimental results discussed above, we briefly survey existing literature closest to our efforts.

Tiny inference on lightweight cores. Albeit rare, existing works demonstrate neural inference on Cortex-M0-class devices. Nyamukuru et al.[36] deploy a gated recurrent unit on a Cortex-M0+ using 8-bit quantization and memory optimizations, achieving 96% accuracy with 6 ms inference time—only 2% below the full-precision baseline. Khatoon et al.[24] use a binary neural network for road anomaly detection, improving inference speed by 25% with just a 2.5% accuracy loss. A 3-layer CNN with 1.7k parameters achieved 95% MNIST accuracy on Cortex-M0+ MCUs using 18–23 KB of program memory, with inference times from 8 to 37 ms [34].

Unlike existing work that applies quantization or pruning post-training, we restructure the architecture to align with MCU constraints. On MNIST, Neuro-C achieves 97% accuracy, above the 95% reported for quantized CNNs [34], while delivering faster inference. Normalized for clock speed, it turns out Neuro-C uses only 6% of the inference time of Cortex-M0/M0+ baselines.

Binary and ternary neural networks. Binary and ternary networks such as BinaryNet [12], XNOR-Net [54] and DoReFaNet [53] reduce compute requirements but rely on batch normalization and retain high-precision layers for stability. They typically use per-layer scaling and focus on large CNNs, not fully connected architectures. Tools such as Larq [16] support efficient inference but assume SIMD/DSP support and exclude MCUs like Cortex-M0; Larq Engine targets 64-bit ARM cores only. Prior BNN deployments on Cortex-M cores exist, but yield high latencies even at high clock settings, such as 216MHz [39].

Existing ternary networks rely on per-layer scaling and batch normalization for convergence. With fixed quantized weights, batch normalization cannot be folded into the model and must be computed at runtime, adding substantial overhead that low-end MCUs cannot afford. We introduce per-neuron scaling to enable stable training without batch normalization, and demonstrate efficient ternary inference on Cortex-M0-class MCUs without any custom engine or specialized hardware features.

TinyML methods for more capable MCUs. While several advanced TinyML approaches achieve impressive results on more capable microcontrollers, they rely on hardware features absent in ultra-low-power platforms like Cortex-M0.

MCUNet [29] demonstrates efficient neural inference through hardware-software co-design, but its TinyEngine requires SIMD acceleration and convolution optimizations (kernel fusion, depthwise operations) available only on MACC-enabled Cortex-M4/M7 platforms. Similarly, FAtRELU [26] exploits activation sparsity through SIMD vectorization, AVX512 instructions, and multicore parallelism that are unavailable on constrained MCUs. DeepShift [15] shares our goal of avoiding multiplication operations but targets GPU-equipped systems, retains dense weight matrices, and achieves only marginal memory savings over `int8` quantization when deployed on MCUs.

These features render such methods fundamentally incompatible with our target platform, making even scaled-down variants non-deployable. Our evaluation, therefore, benchmarks against what is realistically deployable on Cortex-M0 today, and demonstrates up to 90% improvements in latency and memory usage in this setting.

8 CONCLUSION

We demonstrate that building neural architectures directly around hardware constraints, rather than adapting existing models, may improve performance and deployability on ultra-low-power MCUs. We achieve this by restructuring the architecture, specifically, by shifting weights from connections to neurons and using ternary connectivity, which allows accurate models to run in a time-predictable manner within tight latency and memory budgets, where standard designs are not deployable. On a Cortex-M0 MCU, for example, Neuro-C reduces inference latency and program memory usage by up to 90% compared to conventional MLPs with similar accuracy.

ACKNOWLEDGMENTS

We would like to thank our shepherd Zhaoguo Wang and the anonymous reviewers for their invaluable comments. This research was supported by the following project: "BOS: Principles and techniques for body-centered operative systems", the Swedish Foundation for Strategic Research (SSF) grant FUS21-0067.

References

[1] M. Afanasov, N. A. Bhatti, A. Naveed, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proc. of ACM Conf. on Embedded Networked Sensor Systems (SenSys)*. 368–381.

[2] Saad Ahmed, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. The betrayal of constant power× time: Finding the missing joules of transiently-powered computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 97–109.

[3] S. Ahmed, B. Islam, K. S. Yildirim, M. Zimmerling, P. Pawełczak, M. H. Alizai, B. Lucia, L. Mottola, J. Sorber, and J. Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* 67, 3 (2024), 64–73.

[4] Taiwo Samuel Ajani, Agbotiname Lucky Imoize, and Aderemi A Atayero. 2021. An overview of machine learning within embedded and mobile devices–optimizations and applications. *Sensors* 21, 13 (2021), 4412.

[5] E. Alpaydin and Fevzi. Alimoglu. 1996. Pen-Based Recognition of Handwritten Digits. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5MG6K.

[6] Analog Devices. 2025. *MAX78000: AI Microcontroller with Convolutional Neural Network Accelerator*. Available at https://www.analog.com/en/products/max78000.html.

[7] Michael P Andersen and David E Culler. 2014. System design trade-offs in a next-generation embedded wireless platform. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-162* (2014).

[8] ARM Ltd. 2023. *Cortex-M4 Devices Generic User Guide*. ARM. Available at https://developer.arm.com/documentation/dui0553/a/.

[9] ARM Software. 2025. *CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M Processors*. Available at https://arm-software.github.io/CMSIS-NN/latest/index.html.

[10] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. 2020. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821* (2020).

[11] Rei Barjami, Antonio Miele, and Luca Mottola. 2024. Intermittent Inference: Trading a 1% Accuracy Loss for a 1.9 x Throughput Speedup. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. 647–660.

[12] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[13] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.

[14] Edge Impulse, Inc. 2025. *Edge Impulse: The Leading Edge AI Platform*. Available at https://edgeimpulse.com/.

[15] Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. 2021. Deepshift: Towards multiplication-less neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2359–2368.

[16] Lukas Geiger and Plumerai Team. 2020. Larq: An open-source library for training binarized neural networks. *Journal of Open Source Software* 5, 45 (2020), 1746.

[17] Google AI. 2025. *LiteRT: High-Performance On-Device AI Runtime*. Available at https://ai.google.dev/edge/litert.

[18] Rishabh Goyal, Joaquin Vanschoren, Victor Van Acht, and Stephan Nijssen. 2021. Fixed-point quantization of convolutional neural networks for quantized inference on embedded platforms. *arXiv preprint arXiv:2102.02147* (2021).

[19] Prasanna Hambarde, Rachit Varma, and Shivani Jha. 2014. The survey of real time operating system: RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. IEEE, 34–39.

[20] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] Nieves G Hernandez-Gonzalez, Juan Montiel-Caminos, Javier Sosa, and Juan A Montiel-Nelson. 2024. An Edge Computing Application of Fundamental Frequency Extraction for Ocean Currents and Waves. *Sensors* 24, 5 (2024), 1358.

[22] Fatma Karray, Mohamed W Jmal, Alberto Garcia-Ortiz, Mohamed Abid, and Abdulfattah M Obeid. 2018. A comprehensive survey on wireless sensor node hardware platforms. *Computer Networks* 144 (2018), 89–110.

[23] Mehrdad Khani, Pouya Hamadanian, Arash Nasr-Esfahany, and Mohammad Alizadeh. 2021. Real-time video inference on edge

devices via adaptive model streaming. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 4572–4582.

[24] Amna Khatoon, Weixing Wang, Asad Ullah, Limin Li, and Mengfei Wang. 2024. Optimized Binary Neural Networks for Road Anomaly Detection: A TinyML Approach on Edge Devices. *Computers, Materials & Continua* 80, 1 (2024).

[25] Hyung-Sin Kim, Michael P Andersen, Kaifei Chen, Sam Kumar, William J Zhao, Kevin Ma, and David E Culler. 2018. System architecture directions for post-soc/32-bit networked sensors. In *Proceedings of the 16th ACM conference on embedded networked sensor systems*. 264–277.

[26] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. 2020. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *International Conference on Machine Learning*. PMLR, 5533–5543.

[27] Minh Tri Lê, Pierre Wolinski, and Julyan Arbel. 2023. Efficient neural networks for tiny machine learning: A comprehensive review. *arXiv preprint arXiv:2311.11883* (2023).

[28] Fangyuan Lei, Xun Liu, Qingyun Dai, and Bingo Wing-Kuen Ling. 2020. Shallow convolutional neural network for image classification. *SN Applied Sciences* 2, 1 (2020), 97.

[29] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in neural information processing systems* 33 (2020), 11711–11722.

[30] Onel LA López, Osmel M Rosabal, David E Ruiz-Guirola, Prasoon Raghuwanshi, Konstantin Mikhaylov, Lauri Lovén, and Sridhar Iyer. 2023. Energy-sustainable IoT connectivity: Vision, technological enablers, challenges, and future directions. *IEEE Open Journal of the Communications Society* 4 (2023), 2609–2666.

[31] Ioan Lucan Orășan, Ciprian Seiculescu, and Cătălin Daniel Căleanu. 2022. A brief review of deep neural network implementations for ARM cortex-M processor. *Electronics* 11, 16 (2022), 2545.

[32] Andrea Maioli, Kevin Alessandro Quinones, Saad Ahmed, Muhammad Hamad Alizai, and Luca Mottola. 2025. Dynamic Voltage and Frequency Scaling for Intermittent Computing. *ACM Transactions on Sensor Networks* 21, 2 (2025), 1–34.

[33] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. 2020. Mlperf training benchmark. *Proceedings of Machine Learning and Systems* 2 (2020), 336–349.

[34] NXP Semiconductors. 2021. *Inferencing Deep Learning on Cortex M0/M0+ with the eIQ™ Glow Inference Engine.* Technical Report AN13438. NXP Semiconductors. Available at https://www.nxp.com/docs/en/application-note/AN13438.pdf.

[35] NXP Semiconductors. 2025. *Kinetis® K Series: High-Performance Microcontrollers (MCUs) Based on Arm® Cortex®-M4 Core.* Available at https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/k-series-arm-cortex-m4:KINETIS_K_SERIES.

[36] Maria T Nyamukuru and Kofi M Odame. 2020. Tiny eats: Eating detection on a microcontroller. In *2020 IEEE Second Workshop on Machine Learning on Edge in Sensor Systems (SenSys-ML)*. IEEE, 19–23.

[37] Renesas Electronics Corporation. 2025. *AI Accelerator: DRP-AI.* Available at https://www.renesas.com/en/software-tool/ai-accelerator-drp-ai.

[38] Adnan Sabovic, Michiel Aernouts, Dragan Subotic, Jaron Fontaine, Eli De Poorter, and Jeroen Famaey. 2023. Towards energy-aware tinyML on battery-less IoT devices. *Internet of Things* 22 (2023), 100736.

[39] Fouad Sakr, Riccardo Berta, Joseph Doyle, Alessio Capello, Ali Dabbous, Luca Lazzaroni, and Francesco Bellotti. 2024. CBin-NN: an inference engine for Binarized neural networks. *Electronics* 13, 9 (2024), 1624.

[40] Nikolaos Schizas, Aristeidis Karras, Christos Karras, and Spyros Sioutas. 2022. TinyML for ultra-low power AI and large scale IoT deployments: A systematic review. *Future Internet* 14, 12 (2022), 363.

[41] Prateek Shantharama, Akhilesh S Thyagaturu, and Martin Reisslein. 2020. Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies. *IEEE Access* 8 (2020), 132021–132085.

[42] Weining Song, Stefanos Kaxiras, Thiemo Voigt, Yuan Yao, and Luca Mottola. 2024. TaDA: Task Decoupling Architecture for the Battery-less Internet of Things. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. 409–421.

[43] STMicroelectronics. 2025. *STM32C0 Series - Entry-level 32-bit Arm Cortex-M0+ MCUs.* Available at https://www.st.com/en/microcontrollers-microprocessors/stm32c0-series.html.

[44] STMicroelectronics. 2025. *STM32Cube.AI: AI Model Optimizer for STM32 Microcontrollers.* Available at https://stm32ai.st.com/stm32-cube-ai/.

[45] STMicroelectronics. 2025. *STM32F0 Series - 32-bit Arm Cortex-M0 MCUs.* Available at https://www.st.com/en/microcontrollers-microprocessors/stm32f0-series.html.

[46] STMicroelectronics. 2025. *STM32F072RB: Mainstream Arm Cortex-M0 USB Line MCU with 128 Kbytes of Flash Memory, 48 MHz CPU, USB, CAN and CEC Functions.* Available at https://www.st.com/en/microcontrollers-microprocessors/stm32f072rb.html.

[47] STMicroelectronics. 2025. *STM32L0 Series - Ultra-low-power 32-bit Arm Cortex-M0+ MCUs.* Available at https://www.st.com/en/microcontrollers-microprocessors/stm32l0-series.html.

[48] Filip Svoboda, Javier Fernandez-Marques, Edgar Liberis, and Nicholas D Lane. 2022. Deep learning on microcontrollers: A study on deployment costs and challenges. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*. 54–63.

[49] Vasileios Tsoukas, Anargyros Gkogkidis, Eleni Boumpa, and Athanasios Kakarountas. 2024. A Review on the emerging technology of TinyML. *Comput. Surveys* 56, 10 (2024), 1–37.

[50] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 17–32.

[51] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*. 189–203.

[52] Runpeng Yu, Weihao Yu, and Xinchao Wang. 2024. Kan or mlp: A fairer comparison. *arXiv preprint arXiv:2407.16674* (2024).

[53] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[54] Shien Zhu, Luan HK Duong, and Weichen Liu. 2020. XOR-Net: An efficient computation pipeline for binary neural network inference on edge devices. In *2020 IEEE 26th international conference on parallel and distributed systems (ICPADS)*. IEEE, 124–131.