FAST-INF: Ultra-Fast Embedded Intelligence on the Batteryless Edge

Leonardo Lucio Custode
leonardo.custode@unitn.it
University of Trento
Povo, Trento, Italy

Pietro Farina
pietro.farina@studenti.unitn.it
University of Trento
Povo, Trento, Italy

Eren Yıldız
eren.yildiz@ege.edu.tr
Ege University
Izmir, Turkey

Renan Beran Kılıç
renanberan.kilic@unitn.it
University of Trento
Povo, Trento, Italy

Kasım Sinan Yıldırım*
kasimsinan.yildirim@unitn.it
University of Trento
Povo, Trento, Italy

Giovanni Iacca*
giovanni.iacca@unitn.it
University of Trento
Povo, Trento, Italy

Abstract

Batteryless edge devices are extremely resource-constrained compared to traditional mobile platforms. Existing tiny deep neural network (DNN) inference solutions are problematic due to their slow and resource-intensive nature, rendering them unsuitable for batteryless edge devices. To address this problem, we propose a new approach to embedded intelligence, called FAST-INF, which achieves extremely lightweight computation and minimal latency. FAST-INF uses binary tree-based neural networks that are ultra-fast and energy-efficient due to their logarithmic time complexity. Additionally, FAST-INF models can skip the leaf nodes when necessary, further minimizing latency without requiring any modifications to the model or retraining. Moreover, FAST-INF models have significantly lower backup and runtime memory overhead. Our experiments on an MSP430FR5994 platform showed that FAST-INF can achieve ultra-fast and energy-efficient inference (up to 700× speedup and reduced energy) compared to a conventional DNN.

CCS Concepts

• **Computing methodologies** → **Machine learning**; **Neural networks**; • **Computer systems organization** → **Embedded software**.

Keywords

Batteryless embedded systems, fast feedforward networks

ACM Reference Format:
Leonardo Lucio Custode, Pietro Farina, Eren Yıldız, Renan Beran Kılıç, Kasım Sinan Yıldırım, and Giovanni Iacca. 2024. FAST-INF: Ultra-Fast Embedded Intelligence on the Batteryless Edge. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '24), November 4–7, 2024, Hangzhou, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3666025.3699335

*Both authors contributed equally to this research.

SenSys '24, November 4–7, 2024, Hangzhou, China

ACM ISBN 979-8-4007-0697-4/24/11
https://doi.org/10.1145/3666025.3699335

1 Introduction

Today's Internet of Things (IoT) applications require embedded intelligence on resource-constrained edge devices to obtain timely, accurate, energy-efficient, and privacy-preserving inferences based on data sensed *in situ* [82]. However, embedded intelligence is resource-hungry, while most edge devices are battery-powered and need to operate for extended periods without maintenance [1, 2]. Running energy-hungry deep neural network (DNN) models on these devices can rapidly deplete their batteries, reducing their operational time and effectiveness [65, 71, 72, 75]. Thanks to the latest breakthroughs in electronics and energy harvesting, a new generation of edge devices that can operate without batteries is now a reality [62, 69]. Batteryless operation promises embedded intelligence *forever* without maintenance by using exclusively the energy harvested from the ambient [32].

Compared to conventional mobile platforms, batteryless edge devices are *extremely* resource-constrained. For instance, they are built around 16-bit ultra-low-power microcontroller units (MCUs) with a few kB-sized memory [5, 25, 48]. Furthermore, they have tight energy budgets since they rely solely on energy harvested from the environment to charge their tiny capacitors. Due to scarce and transient ambient energy, they quickly consume stored energy and experience frequent power failures, leading to *intermittent computation* [3, 33]. Several recent studies have focused on batteryless intelligence and demonstrated the intermittent execution of tiny DNN models under resource constraints and stringent energy budgets [15, 32, 81]. However, in our view, these studies have two significant problems, as listed below:

P1: Very slow and energy-hungry inference. Batteryless edge devices may face power failures while executing even a single DNN layer, due to their tiny energy storage capacitors [32, 81]. Thus, several charge/compute cycles (power cycles) are required to complete the inference. It is worth mentioning that there is no one-size-fits-all energy storage architecture for DNN inference. Using a capacitor large enough to complete a DNN inference in one power charge cycle will significantly increase charging time. Therefore, this is avoided in energy storage architectures for batteryless systems, since it degrades reactivity to handle time-sensitive events [76, 77]. Furthermore, sensing and preprocessing operations might still interrupt the DNN inference and lead to power failures, even with larger capacitors. At the end of each power cycle, the device backs up the computational state in nonvolatile memory. When it turns

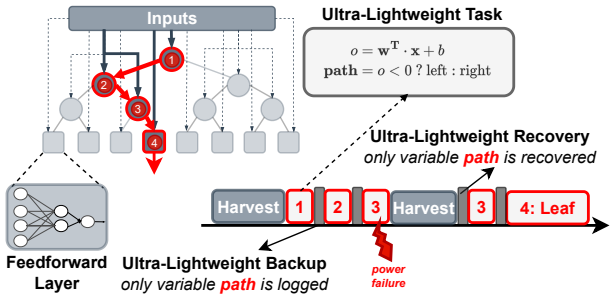

Figure 1: Conceptual scheme of Fast-Inf functioning.

on, it recovers the computational state and resumes the DNN inference. The limited processing capabilities of these devices and overheads due to frequent backup/recovery make compute- and memory-intensive DNN computations extremely slow and energy-inefficient. For instance, running even simple models takes on the order of several seconds to minutes [32, 45, 81], preventing timely responses.

P2: No response to charging time dynamics. Harvested energy can often be unstable due to the sporadic and uncontrollable nature of ambient energy sources. This can lead to longer charging times and increased latency during computations [27, 38, 52]. Unfortunately, DNN models are latency-agnostic as their layers are executed sequentially till the last layer. Some studies have proposed augmenting DNN models with early exit branches [45, 46], which allow the model to terminate early and still provide outputs with reasonable accuracy for the application. However, these solutions are not lightweight since they can introduce significant memory and computational overhead due to the additional parameters of the augmented exit branches‹[28].

Problem statement. Existing DNN-based inference solutions are computationally heavy and energy-hungry (**P1**) and latency-agnostic (**P2**), making them unsuitable for batteryless edge devices. These devices must output accurate results in a few power cycles by consuming only a minimal amount of energy stored in their tiny capacitors [60]. This calls for a new embedded intelligence approach with extremely lightweight computational characteristics, minimal latency, and satisfactory inference accuracy for the application at hand.

Contributions. We introduce Fast-Inf (Fast Inference), a novel ultra-fast embedded intelligence solution specifically tailored to extremely resource-constrained systems. As depicted in Figure 1, Fast-Inf models are binary tree-based neural networks where each inner node of the tree is a single neuron, while the leaves are tiny feedforward layers. An inner node of the tree computes an intermediate output that is then used to decide whether to take the left or the right branch. The inference concludes by executing a feedforward layer when a leaf node is reached. This operation is extremely energy- and time-efficient with significantly lower backup and runtime memory overhead.

To ensure that Fast-Inf models can fit in the extreme edge and executed efficiently, we introduced several novel mechanisms that can summarized as follows. (1) Fast-Inf networks yield a memory/inference time and accuracy trade-off as deeper networks remain computationally lightweight but with an increased memory footprint. To address this challenge, we devised a specific pruning approach, which truncates less important leaves and imposes sparsity to eliminate weights that have minimal impact on accuracy. (2) Furthermore, we introduced truncated inference, a novel approach to make Fast-Inf models adaptive and energy-aware (i.e., aware of the energy harvesting dynamics). (3) Besides, to improve the accuracy, we adopt a custom training procedure that includes regularization in the loss function. (4) Finally, we introduced an energy- and memory-efficient inference engine that enables fast and efficient intermittent inference with minimal overhead on the MSP430FR5994 [40] platform, which is the *de facto* computational platform for batteryless systems.

These contributions make Fast-Inf the first *pure software-based solution* that achieves ultra-fast inference, introduces minimal memory overhead, and offers energy-adaptive latency. In short, Fast-Inf comes with the following features:

(1) *Tiny memory footprint.* Compared to the corresponding DNN architectures, Fast-Inf achieves a significant memory footprint reduction (up to approximately 6× fewer parameters and 4420× less runtime buffer requirements), making Fast-Inf models fit in the small memory of batteryless devices.
(2) *Ultra-fast tiny inference.* Running Fast-Inf models is ultra-fast and energy-efficient due to their logarithmic time complexity. We observed up to approximately 700× speedup and less energy consumption compared to DNNs during our experiments on the MSP430FR5994 MCU.
(3) *Tiny runtime.* Fast-Inf inference engine has approximately 6× less code size and 1000× less runtime overhead compared to the *de facto* inference engine for batteryless devices [32]. Fast-Inf inference tasks have significantly lower backup and runtime memory overheads.
(4) *Adaptable latency.* Fast-Inf can truncate (i.e., skip) the leaves when necessary, minimizing latency without degrading the accuracy significantly, adapting inference to sporadic energy conditions and dynamic constraints.

We are excited to share Fast-Inf with the research community and will release it as an open-source project [21]. We believe that Fast-Inf acts as a solid baseline for future research and leaves a rich design space for future works aimed at further exploration and improvements.

2 Zero-Energy Tiny Inference

Enabling intelligence on the batteryless edge offers intelligence using the "free" ambient energy [32, 33], but comes with several challenges due to extreme resource constraints and intermittent operation. We summarize these challenges and highlight Fast-Inf's unique features to address them.

2.1 Addressing Resource Constraints

Various techniques have been proposed to reduce the parameters of DNN models to make them suitable for memory and computationally constrained embedded devices [10, 14, 35, 50]. These techniques include quantization [31, 34, 47], pruning [35, 36, 54, 55, 57], and separation [10, 32, 73], and have been already implemented in multiple studies targeting batteryless edge devices. For instance, Gobieski et al. [32] utilized pruning, separation, and neural architecture search [56] to obtain a DNN model that can meet the memory and energy requirements of the target device. However, the intermittent

execution of these models still results in substantial overheads, as we explain shortly.

2.2 Challenges of Intermittent Inference

Although energy is an abundant resource, its availability can be affected by various factors, such as the inefficiency of energy harvesting techniques and the erratic nature of ambient energy. These factors, along with the increasing computational demand, often result in energy shortages and power failures [5, 23, 25, 37]. A power failure leads to the loss of the computation state, i.e., a failure typically clears the contents of the CPU registers and the volatile memory. Consequently, the computation returns back to its main entry point when power is restored, but it might not *progress forward*. Furthermore, the re-execution of a code block after a power failure might keep persistent variables (i.e., variables kept in non-volatile memory) in an *inconsistent state* due to Write-After-Read (WAR) dependencies [67]. Several software solutions have been proposed to address these issues [7, 11, 18, 19, 52, 58, 59, 79, 81]. Briefly, these solutions run computations intermittently across multiple power cycles by backing up the computational state in non-volatile memory when power failure is imminent and restoring it when sufficient energy becomes available for resumption.

In this work, we consider the *task-based* model [7, 19, 58, 59, 79], a lightweight intermittent computing approach that requires the computation to be defined as a set of failure-atomic and idempotent tasks that can be safely re-executed upon power failures [19, 58, 59, 79, 80]. Several recent works focused on the task-based implementation of custom DNN workloads and their efficient intermittent execution [15, 32].

Computationally heavy tasks. Gobieski et al. [32] tested a task-based implementation of a DNN for the MNIST dataset [24]. This implementation involves 18 complex computational tasks, with the convolution task alone requiring several hundred thousand multiply-and-accumulate (MAC) operations (e.g., `o+=w*x+b`). Unfortunately, performing these tasks on MCUs with limited processing power on batteryless platforms results in significant delays, and any energy spent is wasted if computation is interrupted by a power failure [32, 81]. While low-power hardware accelerators found in batteryless platforms could speed up the process, they still consume a significant amount of energy and lose computational state in the event of a power failure [15, 32, 49, 53].

Backup and runtime memory overhead. MAC operations executed by DNN tasks have WAR dependencies. To preserve idempotency and enable failure-atomic execution of these tasks, their inputs and outputs are separately maintained in non-volatile memory in a *runtime buffer* (a.k.a. working buffer) [32]. Consequently, the runtime buffer requirement of a layer is the sum of its input and output sizes. Thus, the runtime memory overhead of a DNN model is determined by the layer with the highest input/output requirements. Unfortunately, DNNs introduce large runtime memory overhead due to the large number of (typically, large) layers being used. Besides, DNN tasks need to back up their outputs upon completion. This is needed to avoid losing their computational results due to a power failure, but it introduces significant energy and time overhead. Tasks with larger runtime buffer requirements have considerably larger backup overheads.

Table 1: Prior works on batteryless intelligence.

Solutions	Features
Rehash [7], AdaMICA [4], Camaroptera [25], ImmortalThreads [81], Neuro.ZERO [53], Protean [5], SONIC [32], SoundSieve [60]	Compressed DNN models, slow inference, high energy overhead, no adaptable latency.
HarvNet [46], Zygarde [45], ePerceptive [61], FreeML [28]	Compressed DNN models, slow inference, high energy overhead, adaptable latency via early-exit branches.
FAST-INF (this work)	**tree-based** networks, **ultra-fast** inference, **very-low** energy overhead, **truncated** inference.

Latency Adaptation Overhead. Prior works used early exit branches to adapt inference time considering the available energy [28, 45, 46, 61]. However, these branches introduce memory, backup, and computational overhead w.r.t. a conventional DNN model, since their inputs need to be computed and preserved in non-volatile memory during inference.

2.3 Unique Features of FAST-INF

Table 1 presents a qualitative comparison of the prior works on batteryless intelligence. We identify four main features that make FAST-INF distinct from the current state of the art.

① FAST-INF employs a novel tree-based network architecture that allows conditional execution during inference [9], allowing ultra-fast and energy-efficient inference thanks to its logarithmic time complexity.

② FAST-INF runtime executes its models very efficiently under sporadic ambient energy. Its tasks are lightweight and require just a few bits to store the state of each intermediate node. This allows us to significantly reduce the backup/recovery overhead during intermittent execution.

③ FAST-INF combines structured and unstructured pruning to further reduce the memory footprint of models. It "truncates" the leaves which are mostly responsible for a single class (structured pruning). Moreover, it imposes sparsity while retraining the model, eliminating the weights that have minimal impact on its accuracy (unstructured pruning). This also enables adaptive inference time, allowing the real-time choice of whether to use the pre-computed values for each leaf or compute the actual output.

④ FAST-INF employs a custom loss function during training, which encourages "specialization" of leaves, i.e., it forces the leaves to just concentrate on a subset of the output classes. This allows us to minimize the accuracy drop when pruning FAST-INF models to fit them on batteryless edge devices.

3 FAST-INF on the Extreme Edge

At the core of FAST-INF is the tree-based Fast Feedforward (FFF) network architecture [8], which has a *binary tree* structure depicted in Figure 1. Each inner node of the FAST-INF tree can be seen as a single neuron, while the leaves are small feedforward layers with a single hidden layer.

3.1 Performing Fast Inference

Formally, each inner node i of the tree computes an intermediate output o_i that is then used to decide whether to take the left or the right branch. The intermediate output is computed as $o_i = w_i^T x + b_i$, where o_i is the output of the i-th inner node, x is the input, w is the weight vector of the i-th neuron, and b_i is the bias term of the node. After computing the neuron's output o_i, we move the computation

to the left branch if $o_i < 0$, otherwise, we go to the right branch. This procedure is re-iterated for all the nodes encountered. When a leaf l is reached, we compute the tree output. We do so in two steps: first, we compute the output of a hidden layer in the leaf: $h_l = W_{l,h}^T x + b_{l,h}$ where h_l is the output of the hidden layer for the l-th leaf, $W_{l,h}$ is the weight matrix of the hidden layer of the l-th leaf, and $b_{l,h}$ is the array of biases for layer. Finally, we compute the logits of the model by computing $\hat{y} = W_{l,y}^T x + b_{l,y}$.

3.2 Applicability

As we show in Section 5, Fast-Inf is suitable for problems that can be addressed using fully connected networks (FCNs). This is because this kind of model can be viewed as a tree-based decomposition of FCNs. As a result, Fast-Inf performs comparably to FCNs where the goal is to map instantaneous measurements to a specific class, such as in the case of human activity recognition (HAR) [39], which is a popular embedded application that classifies activities using accelerometer data. On the other hand, for tasks that involve spatiotemporal structure (e.g., audio samples), FCNs typically perform worse than convolutional neural networks (CNNs) in terms of accuracy. However, Fast-Inf can still achieve satisfactory performance on tasks with reasonably small input sizes, such as keyword spotting (KWS) [74]. This is another common embedded application that performs speech recognition to identify a set of target keywords through a microphone.

3.3 Training Fast-Inf Models

The Fast-Inf training process aims to build a tree-based neural network with *minimal depth* that achieves the target accuracy. The training is iterative: it starts with a network of depth 1, trains it, and checks if the target accuracy is achieved. If not, the depth of the network is increased and the training procedure is repeated. This procedure stops when a network with the given target accuracy is obtained.

During training, the output of the model is a weighted sum of the outputs of each leaf, as initially introduced in [30]. Practically, the outputs of the inner nodes are converted to probabilities (of taking the right branch):

$$p_i = \mathbb{P}(right|i) = \sigma(o_i) \tag{1}$$

where σ represents the sigmoid function. Thus, the probabilities of the inner nodes are combined to use them as weights for a leaf l:

$$P_l = \mathbb{P}(path|root). \tag{2}$$

This means that, during training, the output of the model is:

$$\hat{y} = P_0\hat{y}_0 + P_1\hat{y}_1 + ... + P_N\hat{y}_N \tag{3}$$

where N is the number of leaves.

At each training step, $\hat{y}$ is used to compute the loss for the back-propagation as:

$$\mathcal{L} = \mathcal{L}_{xh} + w_{L2} \cdot \mathcal{L}_2. \tag{4}$$

where $\mathcal{L}_{xh}$ is the cross-entropy loss between the logits $\hat{y}$ and the real classes y and $\mathcal{L}_2$ is the norm of the parameters used in the leaves. The $\mathcal{L}_2$ loss on the leaves' parameters pushes unnecessary weights to 0, yielding several advantages: (1) it facilitates compression by means of pruning based on the magnitude of the weights [29, 32], because the unnecessary weights already have very low magnitude; and (2) it leads to learning simpler input-output functions, which perform as few operations as possible on the input data to compute the output. Indirectly, this loss forces the model to learn a more effective tree structure, where each leaf is responsible for the prediction for just a few classes, instead of all of them. The loss function is optimized using Adam [51].

After training, the tree is "discretized" by allowing it to perform inference in logarithmic time w.r.t. the total number of neurons in the leaves. This is done by simply taking the path with the highest likelihood, i.e.:

$$\mathbb{P}(right|i) = \begin{cases} 0 & o_i < 0 \\ 1 & \text{otherwise.} \end{cases} \tag{5}$$

Implementation-wise, this can be seen as a tree traversal: when $o_i < 0$, the computation continues to the left branch, otherwise, it continues to the right branch.

3.4 Boosting Inference Efficiency

Given a binary Fast-Inf with leaf width w, depth d, input size s_i on output size s_o, we can compute:

- the *inference cost* as $\theta(d \cdot s_i + s_i \cdot w + w \cdot s_o)$;
- the *memory footprint* as $\theta(2^{d-1} \cdot s_i + 2^d \cdot s_i \cdot w + 2^d \cdot w \cdot s_o)$.

Intuitively, deeper networks lead to an increased memory footprint and inference cost. As mentioned earlier, to remedy this, Fast-Inf combines structured and unstructured pruning by ① truncating less important leaves (structured pruning) and ② imposing sparsity to eliminate weights that have minimal impact on accuracy (unstructured pruning).

3.4.1 Leaf Truncation. In Fast-Inf, the leaves carry out a significant portion of the computation and hold most of the weights. As such, it is crucial to determine when a leaf's computation is needed and when it can be reduced. To address this aspect, we introduce a leaf truncation mechanism, which can be seen as a form of *structured pruning*. Our approach takes into account the *a priori* probabilities of each class for each given leaf. The underlying assumption is simple: if the *a priori* probability of the most likely class in a given leaf is higher than a threshold, then we "cut" the hidden layers in the leaf and replace them with constant logits. Formally, if:

$$\max_i(\mathbb{P}(i|l)) > \xi \tag{6}$$

then we can set:

$$\hat{y}_l^* = \{\mathbb{P}(0|l), \ldots, \mathbb{P}(n|l)\} \tag{7}$$

where ξ is a hyperparameter, and n is the number of classes.

Note that the introduction of this truncation mechanism introduces a dual advantage. In fact, one may either:

(1) eliminate the truncated leaves, significantly reducing both the inference time and the memory consumption of the model;
(2) or, keep the truncated leaves in the model and choose, at runtime, whether to use the pre-computed values for the leaf or to compute the actual output of the leaf.

It is important to stress once again that the introduction of this mechanism is made possible by the introduction of the $\mathcal{L}_2$ loss during training. In fact, as specified in the previous subsection, this loss encourages learning simpler input-output models in the leaves

and, as a consequence, learning better structure for the trees, so that each leaf has to predict just a few classes from the whole set of classes. Thus, when a given leaf "receives" (i.e., is queried for) samples from a given class, it can be replaced with a "pre-computed" prediction, which uses the *a priori* probability for each class.

Note that this loss is needed because of the way the trees are trained. Since during training all leaves participate in each prediction, their weights are usually updated to increase the performance of the whole model (i.e., the weighted sum of the leaves). By using an $\mathcal{L}_2$ loss on the leaves' parameters, we enforce a "specialization" of the leaves, which in turn allows us to learn tree structures that try to "route" a given sample to the leaf specialized for its corresponding class.

3.4.2 Depth-by-depth Compression by Imposing Sparsity. The leaf truncation mechanism can greatly help in reducing both the memory cost and the inference cost. However, not always it is possible to truncate leaves, which may limit the application of these systems in resource-constrained devices. Moreover, as model depth grows, the child nodes also introduce non-negligible memory overhead. For this reason, we introduce another specific compression mechanism that can further reduce the amount of memory used by FAST-INF models. This approach, which can be seen as a form of *unstructured pruning*, works as follows.

Compression Iteration. Our compression method employs an iterative pruning technique, which compresses the entire tree in a depth-wise manner until it can fit into the targeted device's memory. The procedure initiates with the largest leaves (i.e., the nodes at depth l), where a certain percentage of elements of their weight matrix $\mathcal{W}_l$ are set to 0. The model is subsequently retrained, and this process is repeated until the accuracy drop is insignificant. Our algorithm then moves to the largest nodes at the next depth $l-1$, sparsifying the weight matrix $\mathcal{W}_{l-1}$, and so on.

Pruning Weights. At each epoch, our algorithm selects a batch of samples from the dataset and applies *projected gradient descent* (PGD) to update the weights and remove the unnecessary ones. This involves two steps. ① Firstly, we calculate gradients in relation to the weights of the nodes at depth l, which we denote as $\mathcal{W}_l$. We then update the weight matrix by moving in the direction of the negative gradient. ② Secondly, we sparsify the weight matrix using a hard thresholding procedure that sets $\mathcal{S}\%$ of the weights with smaller magnitudes to 0. These steps can be formalized as:

$$\mathcal{W}_l \leftarrow \mathcal{W}_l - \eta \sum_i \nabla_{\mathcal{W}_l} \mathcal{L}_i \tag{8}$$

$$\mathcal{W}_l \leftarrow \text{hardThreshold}(\mathcal{S}, \mathcal{W}_l). \tag{9}$$

Note that the weights set to 0 in an epoch might re-appear in the next epoch. Through multiple iterations, the weight matrix tends to stabilize (keeping only some specific non-zero weights), so that $\mathcal{W}_l$ satisfies the given sparsity without degrading the accuracy of the FAST-INF model significantly.

4 Implementation

We implemented the FAST-INF model training and pruning in Python using the PyTorch framework. Moreover, we implemented a tiny task-based inference engine to run FAST-INF models intermittently. The engine can adapt the inference latency by skipping leaf computations when power failure is imminent and provide an immediate value that represents an approximation of these computations. We selected the MSP430FR5994 [40] MCU, the state-of-the-art microcontroller used in batteryless platforms, as the target hardware platform for our experiments. This MCU offers 256 kB of FRAM [43] and 8 kB of SRAM memory. The FRAM stores the FAST-INF inference code, the parameters of the FAST-INF model, the computational state to be logged, and the runtime buffer used to execute the FAST-INF model intermittently.

4.1 Leaf Truncation and Compression

After training, to perform leaf truncation, we execute a script that reloads the dataset and checks the *a priori* probabilities as discussed earlier. We obtain a wrapped version of the model that allows us to have both the pre-computed values for the simplified leaves and the parameters of the truncated leaves, enabling us to choose whether to keep the truncated leaves or not. Then, we apply unstructured pruning described in Section 3.4.2. We start the compression process by setting the sparsity to 45%, i.e., $\mathcal{S} = 0.45$, and gradually increase it to reach the required size. We monitor the accuracy during retraining and decrease sparsity if the accuracy drop exceeds 3-5%, to prevent over-pruning. To address overfitting during retraining, we use the $\mathcal{L}_2$ regularization term.

4.2 Model Representation

After truncation and compression, we use a script that automatically translates the parameters of the FAST-INF model into C arrays and stores them in a single header file. This header file is used by the intermittent execution runtime, explained in Section 4.3, which navigates the tree, reaches the leaves, and computes the output in a power failure-resilient manner. The arrays in this header file include the weights and biases of the inner nodes of the tree (`node` array), the hidden layers of the leaves (`hidden` array), and the output layers of the leaves (`output` array). The FAST-INF utilizes the Compressed Sparse Row (CSR) representation, which creates sparse arrays by keeping only the non-zero values and their corresponding indices [35], which is computationally efficient when performing arithmetic operations.

We also extract a `fast_inference` array that contains the pre-computed values for each leaf. The advantage of this representation is twofold. ① We can obtain the smallest model architecture by only keeping the `fast_inference` array and excluding the parameters of the truncated leaves (`hidden` and `output` arrays). This speeds up the computation and reduces memory consumption, making it ideal for *extremely* resource-constrained systems. ② We can keep all the arrays, allowing the runtime choice between using pre-computed values or re-computing the outputs for the current input *adaptively* considering the energy and latency requirements, as mentioned earlier in Section 3.4.1.

4.3 Intermittent Inference Engine

The FAST-INF inference engine utilizes a task-based programming model [7, 19, 79] to execute FAST-INF models in a power-failure-resilient manner. Tasks have lightweight computational characteristics and minimal backup overhead. FAST-INF preserves a structure of type `model_t` in non-volatile memory, which encapsulates the binary tree-based neural network by maintaining the tree depth,

the leaf shapes, and pointers to the arrays that hold the model parameters.

4.3.1 Inference Engine. The Fast-Inf inference engine includes four core tasks: `runModel_t`, `neuron_t`, `tree_t`, and `leaf_t`, which are described below.

Starting inference. The `runModel_t` task is the entry point of the inference procedure. The application invokes this task by providing a pointer to the `model_t` structure that holds the Fast-Inf model.

Computations. The `neuron_t` task is the main computational task that carries out the essential dot product operation by performing MAC operations. Since these tasks have all-or-nothing semantics, their computational progress and the energy used are lost if they are interrupted by power failures [81]. In order to prevent this issue, we incorporated *loop continuation* in our implementation, introduced by Gobieski et al. [32]. This feature enables the computation to resume from the latest iteration in the loop nests when the interrupted task is restarted.

Traversing the model. The `tree_t` task navigates the tree by iteratively invoking the `neuron_t` task to select the next child node on the path. When a leaf node is reached, the `leaf_t` task comes into play, executing the feedforward model by iteratively invoking the `neuron_t` task.

Result. Upon completion, Fast-Inf writes the output to the designated address in nonvolatile memory.

```
task_t leaf_t{
 if(fast){ // if flag is set, then fast inference
   result = fast_inference[node_id];
   next_task(tree_t);
 }else{
   ... // run the feedforward layer
 }
}
```

Figure 2: Pseudocode of the truncated inference.

4.3.2 Fast and Adaptive Inference Mode. Fast-Inf maintains a Boolean flag `fast` to adapt the inference latency in a simple but effective way. The `leaf_t` task, whose pseudocode is presented in Figure 2, checks this flag. In case the `fast` flag is set, the task sets the `result` variable by using the pre-calculated value in the `fast_inference` array and finalizes the inference by setting `tree_t` as the next task in the control flow. Otherwise, the task executes the feedforward model by invoking the necessary tasks. It is worth mentioning that `next_task` is used to set the next task in the task-based control flow, which is a keyword forming the basic building blocks of the task-based programming model in intermittent computing [79]. The `fast` flag can be set by an interrupt service routine or explicitly by the application. Potential triggers for the fast inference may occur when a deadline or power failure is imminent. This adaptable method allows for the selection of an optimal inference approach that can be modified to suit different runtime conditions.

5 Evaluation

We conducted the first part of our experiments on our testbed using the MSP430FR5994 launchpad. This microcontroller is the *de facto* computational platform for batteryless systems since it utilizes an embedded FRAM as non-volatile memory. Compared to Flash memory, FRAM has fast read/write times (roughly 1000x faster), nearly unlimited read/write cycles (100 trillion), and operates with low-power consumption [42]. Furthermore, since FRAM is an embedded component of the MSP430FR5994's internal architecture, the processor can access it energy-efficiently using the optimized interconnecting bus, also making the backup and recovery operations energy-efficient. We present the results obtained with MSP430FR5994 in Section 5.1. In the second part of our evaluations, as detailed in Section 5.2 and Section 5.3, we examine Fast-Inf's accuracy in more detail, particularly studying the impact of model structure, adaptation, and compression on accuracy through simulations. Finally, in Section 5.4, we compare Fast-Inf to established models from the state-of-the-art.

Datasets. We evaluated Fast-Inf by testing it on three datasets, which have been selected for being representative of three different application domains. The first dataset is MNIST [24], which can be seen as an instance of a simple, yet plausible image-based application. Secondly, we considered the Google Speech Commands v2 dataset with 10 classes for Keyword Spotting (KWS) [74] as an instance of an audio-based application. For KWS, we used as inputs to the model 13 Mel-Frequency Cepstral Coefficients (MFCCs), obtained by applying Short-Time Fourier Transformation with a window size of 25ms and a hop size of 16ms, which gives 793 (61×13) features for a 1-second sample. Finally, we utilized HAR [39], which classifies activities using accelerometer data, as an example of a wearable application. For each dataset, we used the training/test splits proposed in the original papers [24, 39, 74], to prevent data leaking.

Table 2: The structures of CNNs and Fast-Inf models in evaluations. "D" stands for depth, "L" stands for leaf width, "C" and "F" indicate convolutional and fully connected layers, respectively (with their size).

Dataset	MNIST		KWS [74]		HAR [39]	
Model	Fast-Inf	CNN	Fast-Inf	CNN	Fast-Inf	CNN
Layers	D:3 L:4	C:2 F:3	D:4 L:4	C:4 F:3	D:3 L:16	C:6 F:5
Parameters (kilo)	31	444	335	102	42	413

Baseline Model Structures. Table 2 presents the model structures considered in our evaluations. Our CNNs are similar to those presented in [5, 28, 32], targeting constrained embedded systems. We selected the depths of the Fast-Inf models and leaf widths to achieve comparable accuracy to these CNN models. Since Fast-Inf can be viewed as a tree-based decomposition of FCNs [8], we also included FCNs in our evaluations (not shown in Table 2). These FCNs have the same number of parameters as Fast-Inf models, i.e., the hidden layer width of the FCN model is set equal to the training width ($2^d w$) of the corresponding Fast-Inf model. We excluded binary neural networks and decision trees from our testbed experiments due to their poor accuracy, see our results presented in Section 5.4.

Obtaining Ultra-Tiny Models. We compressed all our models (CNNs, Fast-Inf models, and FCNs) using the unstructured pruning approach described in Section 3.4.2 to fit them in the memory of our target platform: as mentioned earlier, the MSP430FR5994 MCU

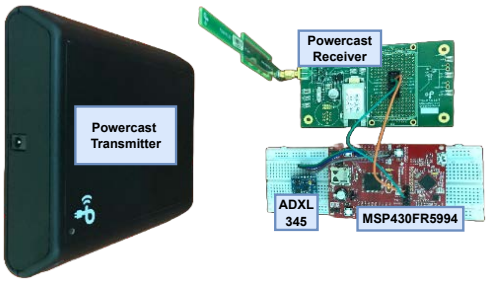

Figure 3: Energy harvesting hardware setup.

has only 256 kB FRAM to store model parameters and runtime buffers. We used these compressed versions in our evaluations. It is worth mentioning that the accuracies of the compressed models were similar to the model accuracies we report in Figure 12, see Section 5.4.

5.1 Testbed Experiments

We ran the compressed models on an MSP430FR5994 microcontroller (MCU) and compared their performances using various metrics. The MCU clock frequency was set to 1 MHz. To ensure consistency, we simulated controlled power failures by triggering the brown-out-reset mechanism of the MCU at regular intervals ranging from 5 to 20 ms. For real energy harvesting-based experiments, we utilized the Powercast TX91501-3W-ID power transmitter [64] and the P2110-EVB harvester [63] equipped with a 1mF energy storage supercapacitor. Our setup is presented in Figure 3.

The P2110-EVB has an output pin that supplies regulated 3.3V to power our system. It operates in a way that when the capacitor voltage reaches 1.25V, the output pin provides 3.3V, and when the voltage drops to 1.05V, the output decreases to 0V, causing a power failure. Consequently, its onboard 1mF capacitor stores approximately 230μJ of energy in each power cycle. We used the logic analyzer mode of Analog Discovery 2 [26] and a GPIO pin of MSP430FR5994 set high at the end of the inference to measure the inference time. We used the Energy Trace tool [41] to measure the consumed energy during inference.

Model Implementations. We implemented the CNN models in Table 2 using Sonic [32], the de facto DNN-based intermittent inference engine for the extreme edge based on the Alpaca [58] task-based model. We implemented the FAST-INF and FCN models using the task-based FAST-INF inference engine presented in Section 4.3.

Evaluation Metrics. We considered the following evaluation metrics. (1) *Runtime Overhead*, i.e., the time overhead introduced by the intermittent computing engine to execute models intermittently, in particular, due to the backup/recovery operations. (2) *Inference Latency* and (3) *Energy Consumption*, which represent the time and energy required to complete the whole model execution. (4) *Memory Overhead*, which indicates the memory requirements for storing the inference engine code, the model parameters, and the memory space for maintaining intermediate computational results.

5.1.1 Latency and Energy Consumption. We evaluated the execution time and energy consumption of FAST-INF and the baseline models during both continuous (*Cont*) and intermittent execution with controlled power failures (*Int*). Our results are presented in Figure 4 and Figure 5, which show, respectively, the latency and energy requirements of the baselines normalized w.r.t. those of FAST-INF models. Additionally, we present the absolute execution time and energy consumption values in Table 3 and Table 4. We observe that FAST-INF models executed by our runtime can significantly reduce inference latency and energy consumption. For instance, concerning the KWS application, FAST-INF reduced the inference latency approximately by **607**× during intermittent execution compared to the corresponding CNN model executed by Sonic runtime. While this result may seem counterintuitive (due to the fact that our FAST-INF network has 3× more parameters than the CNN) it is important to note that, in FAST-INF, each parameter is used only once during inference, while a CNN reuses some parameters more than once (namely, the convolutional filters are applied to all the patches of the image). Besides, considering the total energy that can be stored in our setup, which is 230μJ, FAST-INF can complete the inference in a few power cycles. Similarly, concerning the HAR application, FAST-INF reduced the inference latency by **7.27**× during intermittent execution compared to the corresponding FCN model. These observations are also aligned with the measured energy consumption.

Table 3: Total execution time in seconds during the continuous and intermittent execution.

Dataset	Continuous			Intermittent		
	CNN	FCN	FAST-INF	CNN	FCN	FAST-INF
MNIST	93.7	1.9	0.4	108.3	2.1	0.42
HAR	74.6	2.1	0.31	83.4	2.4	0.33
KWS	184.6	1.3	0.32	218.7	1.4	0.36

Table 4: Total energy consumption during the continuous and intermittent execution in mJ.

Dataset	Continuous			Intermittent		
	CNN	FCN	FAST-INF	CNN	FCN	FAST-INF
MNIST	143.7	2.0	0.62	207.9	3.78	0.76
HAR	135.8	3.27	0.59	152.3	4.11	0.71
KWS	355.2	3.27	0.5	419.9	4.11	0.71

Table 5: Detailed execution time profile of the nodes and leaves in FAST-INF models.

Dataset	Nodes (ms)			Leaf (ms)		
	Min.	Avg.	Max.	Min.	Avg.	Max.
MNIST	55.8	70.3	85.5	68.2	148.0	234.9
HAR	27.0	29.4	31.9	141.9	222.1	285.5
KWS	3.0	29.6	87.2	9.3	89..4	299.0

Latency of the Nodes and Leaves. The execution time of each branch in a FAST-INF model is affected by the sparsity of the weights of the nodes and the leaves since sparse matrix multiplications in software take varying amounts of time. Table 5 shows the minimum, maximum, and average computation times for the leaves and nodes in FAST-INF models. Depending on the sparsity level, the compressed model representation might introduce a negligible computational overhead for the inner nodes due to the cost of indirection introduced by CSR representation mentioned in Section 4.2. However, this cost is significantly compensated by the skipped multiplications thanks to the sparsity. We observed that compressing leaves significantly reduces their computation time, e.g., up to **4**× for MNIST.

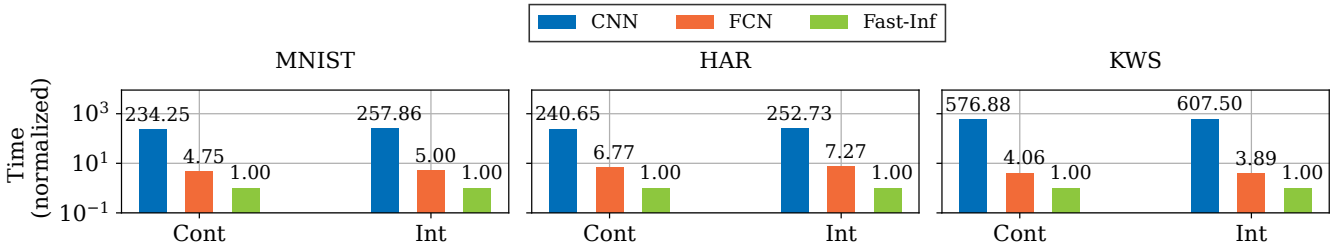

Figure 4: Testbed Experiments: Total execution time during the continuous and intermittent execution. Data are normalized w.r.t. the execution time of Fast-Inf models.

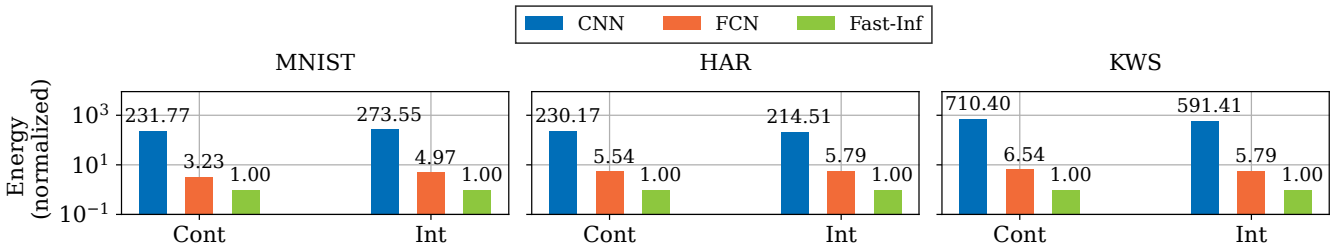

Figure 5: Testbed Experiments: Total energy consumption during the continuous and intermittent execution. Data are normalized w.r.t. the energy consumption of Fast-Inf models.

Table 6: Time overheads (sec) and number of tasks of Sonic and Fast-Inf inference engines.

Dataset	Fast-Inf			Sonic		
	Pure C	Runtime Ov.	Tasks	Pure C	Runtime Ov.	Tasks
MNIST	0.22	0.08 (275×)	5	74	22	22
HAR	0.24	0.07 (257×)	5	56	18	22
KWS	0.18	0.06 (1020×)	5	123	61	22

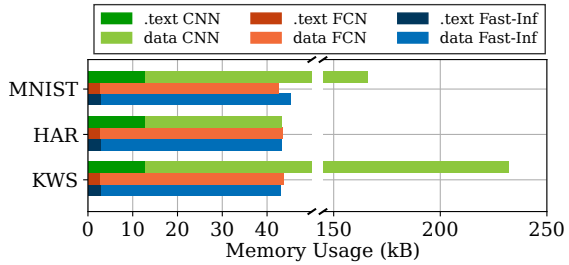

Figure 6: Memory overhead (in kB) of Fast-Inf and compressed base-line models.

Inference Engine Overheads. To understand the backup/restore overhead introduced by the Sonic and Fast-Inf inference engines, we implemented a "Pure C" version of the CNN, FCN, and Fast-Inf models and ran them continuously without any power failures. We subtracted the execution time of the Pure C versions from the execution time of the Fast-Inf and Sonic models to obtain the runtime overhead introduced to run them intermittently, as presented in Table 6 under the "Runtime Ov." column. Thanks to the energy-efficient and lightweight characteristics of Fast-Inf models, the Fast-Inf inference engine has a minimal runtime overhead that is up to **1020**× smaller than that of the Sonic. This is also due to the fewer number of tasks required to be implemented to run the models, i.e., the Fast-Inf requires only 5 tasks, while the Sonic requires 22 tasks to be implemented.

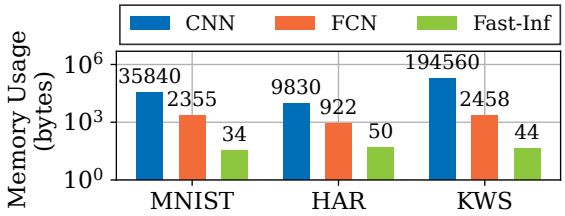

Figure 7: Runtime buffer requirements.

5.1.2 Memory Footprint and Runtime Memory Requirements. We analyzed the `.text` and `.data` segments of the target binary to asses the memory footprint of the models and summarize our measurements in Figure 6. Compared to the Sonic inference engine, the Fast-Inf inference engine, designed to operate with only 5 tasks, significantly reduces the code size by up to **5**×, bringing it down to about 2.5 kB. Besides, the Sonic inference engine requires storing intermediate results of the computation in addition to the model parameters, since each layer's output serves as the input for the next layer. Therefore, Sonic requires a buffer that must be large enough to accommodate the input and output activations of the largest layer. Contrarily, Fast-Inf tasks are minimal and their runtime buffer requirements are extremely small as explained in Section 4.3. As shown in Figure 7, Fast-Inf reduces the runtime buffer requirement by up to **4420**×.

5.1.3 Fast-Inf Adaptive Inference. The Fast-Inf inference engine can employ adaptive execution by skipping the leaves (as mentioned in Section 3.4.1), which can further reduce the execution time. We evaluatedFast-Inf adaptive inference in our testbed shown in Figure 3, using a Powercast transmitter and receiver. We implemented

an activity-tracking application using the HAR model. The main application loop consists of sampling the ADXL45 accelerometer every 1 second and performing activity classification by executing the Fast-Inf inference. If a power failure occurs during inference, Fast-Inf skips the leaf after power recovery by applying leaf truncation. If no power failure occurs, the relevant leaf of the model is executed, and the system moves on to the next loop iteration. We deployed our board at different distances from the energy transmitter to increase the rate of power outages, forcing Fast-Inf to skip leaves and employ truncated inference. We observed that Fast-Inf engine executed almost 20% of the inference iterations by employing truncated inference when we placed our board 1.5 meters from the transmitter. Thanks to the truncated inference, Fast-Inf could boost its inference speed and adapt to different energy conditions by providing a fast inference response. Section 5.2 presents more details on the impact of leaf truncation and its hyperparameter ξ on inference accuracy.

Summary of Testbed Evaluations. Fast-Inf can speed up the intermittent inference and reduce its energy requirements by **two orders of magnitude** with comparable accuracy to state-of-the-art models and significantly reduced runtime memory requirements.

5.2 Fast-Inf Network Structure and Adaptation

In this section, we will explore how the accuracy of the Fast-Inf is impacted by the structural characteristics of the tree it's built on (i.e., the depth of the tree and the width of the leaves) and their adaptive execution through leaf truncation.

5.2.1 Impact of the Tree Structure. The structural characteristics of the tree have a significant impact on the accuracy of the Fast-Inf model. Recall that, by depth, we refer to the number of maximum nodes in the path from the root to a leaf, while by width, we refer to the number of hidden neurons in each leaf. In Figure 8, we observe the following phenomena: (1) in most cases, larger models (in both depth and width) perform better than smaller models, except for KWS, where we observe that there is a peak after which accuracy starts to decay; and (2) when the width of the leaves is too small (e.g., 4, 8 in MNIST, 4 in HAR, and all in KWS), having deeper trees does not help in achieving better performance. Thus, there is no globally optimal choice for the depth and width parameters, but these are task-dependent hyperparameters, as often happens in machine learning models [78]. In our experiments, we selected the depth and width of our models by performing several simulations to evaluate their accuracy and memory requirements after compression.

5.2.2 Impact of the Adaptive Leaf Truncation Leaf truncation is a form of structured pruning that helps to adapt inference computations concerning the available energy. As explained in Section 3.4.1, if the a priori probability of the most likely class in a given leaf is higher than a threshold ξ, then we skip the hidden layers in the leaf and replace them with constant logits. This helps with reducing the computational complexity of the inference by sacrificing its accuracy. Essentially, depending on the ξ parameter, it can draw a trade-off between accuracy and inference speed and energy requirements. A small ξ truncates a significant portion of the leaves (potentially, all of them). On the other hand, using large ξ can potentially prevent the truncation of any leaf, resulting in an unchanged model.

Figure 9 shows how ξ affects performance of Fast-Inf networks on the MNIST dataset, for 3 different depths and 4 different widths. Note that ξ allows us to navigate the trade-off between test accuracy and inference time by specifying the minimum fraction of samples that belong to the majority class for the leaf to be replaced with a constant choice of that class. Intuitively, the lower ξ, the less accurate the classification will be (and the cheaper the inference). We observe the following phenomena: when $\xi \in [0.7, 1]$, there is no significant loss in performance, meaning that we are trading inference speed for better accuracy; when $\xi \in [0.2, 0.7)$, the accuracy can vary significantly depending on the depth and width of the model, meaning that we are giving more importance to inference speed than to accuracy. Finally, when $\xi \in [0, 0.2)$, the accuracy tends to reach the minimum value for all the configurations. The fact that we can safely remove some of the leaves without significantly affecting performance may indicate that there is an imbalance in the frequency of use associated to each leaf. Further investigation highlighted that, in some cases, the usage rate of the most used leaf can be up to 60%, which makes these leaves become the bottleneck for inference time.

Finally, our choice of $\xi = 0.7$ is motivated by the fact that, as shown in Figure 9, this value represents a good trade-off between the number of truncated leaves and performance.

5.3 Fast-Inf Model Compression

Truncation alone can significantly improve the expected inference speed of the resulting models, but still, the problem of fitting these models in a tiny device's memory exists. To this end, the depth-by-depth compression algorithm presented in Section 3.4.2 becomes crucial.

5.3.1 Evaluation of Depth-by-depth Compression We evaluate the performance of our compression algorithm, using the accuracy concerning the model's size reduction as the main evaluation metric. For this analysis, we select a Fast-Inf model with depth 4 and leaf width 8 and start the compression by defining the target size as 32 kB. The algorithm increases, at each iteration, the sparsity constraint to compress the model, and fine-tunes the model accuracy to reduce the accuracy degradation. In Figure 10, we display the accuracy at each compression iteration. It can be seen how this approach maintains, during compression, stable accuracy values up to significantly reduced model size. In short, our unstructured pruning approach can significantly reduce model size (up to 90%) without significant losses in performance, while more aggressive compression can be achieved by trading off accuracy for smaller models.

5.3.2 Optimal Models via Combined Compression and Truncation. To further reduce energy consumption, memory usage, and inference time, we can utilize leaf truncation for model compression. Essentially, instead of storing the parameters of the hidden layers in the leaves, we can just keep a truncated model in memory by storing only the constant logits. Therefore, we combined leaf truncation and pruning, to achieve maximum compression. These two mechanisms are complementary and allow for massive savings in

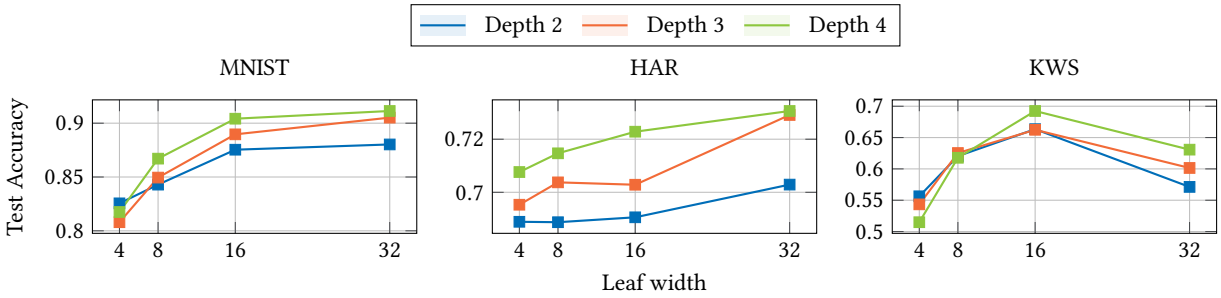

Figure 8: Accuracy vs leaf width of Fast-Inf models.

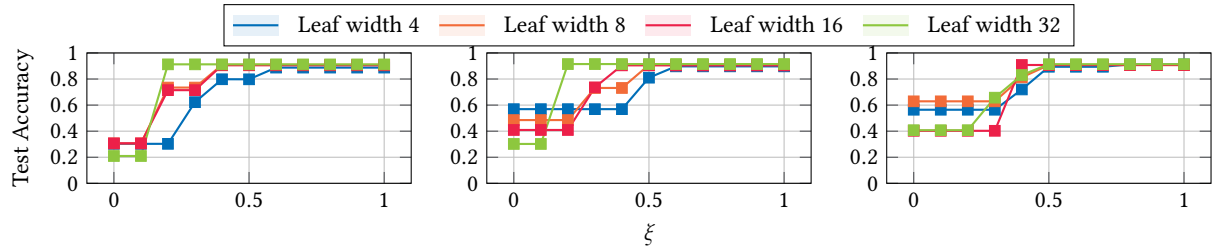

Figure 9: Accuracy for various values of ξ at depth 2 (left), 3 (center), and 4 (right), computed on MNIST.

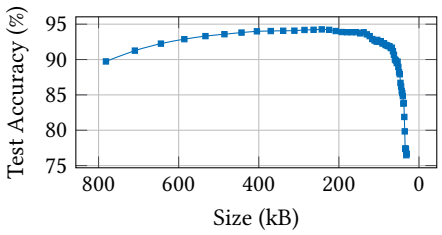

Figure 10: Gradual compression of an MNIST model: accuracy is stable up to a significant reduction in size.

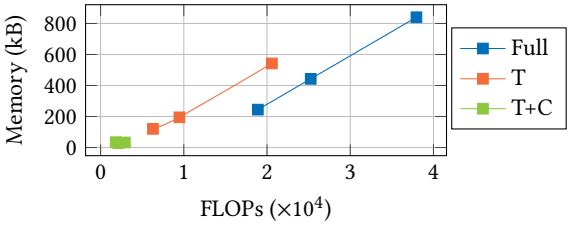

Figure 11: Memory size vs FLOPs for Fast-Inf models with depth 4 on the MNIST dataset. "Full" stands for the full Fast-Inf model, "T" stands for the model after truncation, and "T+C" stands for the model after truncation and compression.

terms of both FLOPs and memory. We applied them simultaneously to Fast-Inf models of depth 4 on the MNIST dataset, as shown by the "T+C" points in Figure 11. We could successfully compress the models to the 7.5% of the original size, without any significant loss in performance.

5.4 Fast-Inf versus SOTA Models

In this section, we compare Fast-Inf to well-established models from the state-of-the-art. We included CNN-based and FCN-based networks as well as binary fully-connected neural networks (BNNs) and decision trees (DTs). We chose to include BNNs because they are

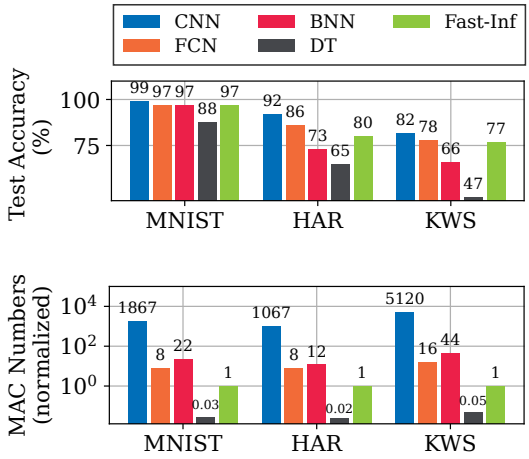

Figure 12: Comparison of Fast-Inf against baseline models with an approximate model size of 60 kB.

also an extremely energy and memory-efficient class of DNNs [16, 17, 66, 68]. Our BNNs have the same architecture as FCNs. We included DTs [13] since they are similar to Fast-Inf in terms of their structure, even though DTs have much simpler comparisons in the inner node (they compare variables to constants instead of comparing linear combinations of the inputs to constants). To train them, we used the Gini impurity as the splitting criterion. We set the minimum number of samples to create a new split to 2.

For fairness, we considered **same-size models** and compared their accuracy. We compressed all models to obtain an approximate model size of 60 kB. For DTs, we have an exception since fixing the size is not feasible with standard DT induction algorithms. Therefore sizes of our DTs vary from approximately 16kB to 47kB. Figure 12 presents our results, in which the top plot presents the

Table 7: Energy consumption and execution time of the total number of MAC operations in mJ and seconds, respectively.

Dataset	CNN		FCN		BNN		DT		Fast-Inf	
	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
MNIST	186.4	159.49	2.98	2.55	11.01	9.41	0.0026	0.0022	0.40	0.341
HAR	106.54	91.14	2.98	2.55	6.26	5.35	0.0076	0.0066	0.40	0.341
KWS	255.69	218.73	3.19	2.73	11.25	9.62	0.0024	0.0021	0.19	0.171

model accuracy. The bottom plot presents instead the number of MAC operations in each model, in order to highlight the models' computational complexity. Furthermore, we also present the models' approximate energy and latency requirements in Table 7, to understand how such models can be sustained by our energy harvesting setup. Note that we did not perform end-to-end hardware measurements to calculate these values, but we approximated them by measuring the time and energy cost of a single MAC operation on MSP430FR5994, which are 45,57 μs and 53,27 nJ, respectively.

Our plots concerning CNNs and FCNs are consistent with the results reported in Section 5.1. Fast-Inf showed comparable performance w.r.t. FCNs while being significantly energy-efficient. On the other hand, Fast-Inf (and FCNs, as well) performs worse than CNNs with an average drop in accuracy of about 6% while being about an order of magnitude energy-efficient (i.e., faster), on average.

Besides, same-size BNN models performed poorly compared to Fast-Inf, especially for more challenging tasks (compared to MNIST) such as HAR and KWS. For such tasks, the network needs more capacity (computations/parameters/depth) to recover the performance loss introduced by the binarized weights. Therefore, in our case, getting a network with low size and high performance was not feasible. Besides being more accurate, Fast-Inf has relatively up to 13× better energy consumption compared to BNNs.

Concerning DTs, their energy consumption was significantly lower not only due to their inherently lightweight characteristics but also because they were relatively smaller in size. However, DTs performed even worse compared to other models. This is because DTs use single variables in the inner nodes, while Fast-Inf uses linear combinations of the inputs, leading to significantly higher expressivity.

Considering the energy harvesting capabilities of our testbed setup, a power cycle corresponds to 230 μJ and Fast-Inf requires approximately 2–3 power cycles to perform a single inference. Therefore, a batteryless edge can easily sustain this energy consumption level for inference. This is a significant improvement considering SOTA models that require at least an order of magnitude more power cycles to complete an inference, justifying the ultra-lightweight characteristics of Fast-Inf.

5.4.1 Detailed Comparison Against Vanilla FFFs. Since FFF networks [8] are the basic building block for Fast-Inf, we made a head-to-head comparison to highlight Fast-Inf's advantages over the vanilla approach. In Figure 13, we compare the best FFF networks obtained with the Fast-Inf models used in Figure 12. We observe that Fast-Inf achieves comparable or better accuracy while reducing inference time. In our opinion, this is because the ℓ^2 loss facilitates the specialization of the leaves, which allows for truncation and for a better decomposition of the input space than FFFs.

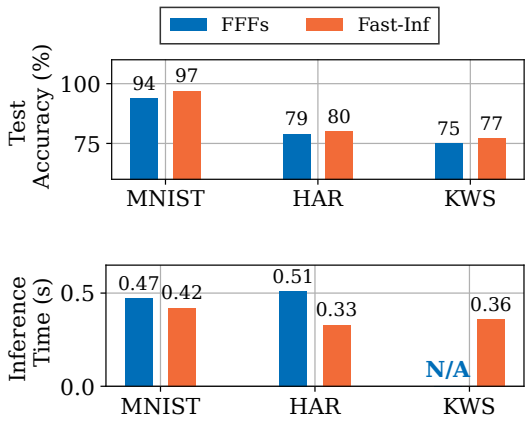

Figure 13: Comparison of Fast-Inf's 60kB models against the best FFF models obtained. N/A means that we could not fit the models in the device's memory.

Summary of Performance Evaluations. Fast-Inf can achieve comparable performance w.r.t. state-of-the-art techniques while greatly reducing inference time, energy consumption, and memory requirements.

6 Discussion and Future Works

Even though Fast-Inf brought many benefits for the embedded intelligence on the extreme edge, our experience has also brought to light that there remains ample room for exploration and advancement regarding this kind of networks.

Decision Trees. While Fast-Inf produces models that can be seen as decision trees, it is important to note that each inner node of the corresponding tree does not compute a simple comparison operation (e.g. comparing a variable to a constant). Instead, it compares a linear combination of the inputs to a threshold (often referred to as an *oblique split*). This allows for trees that are significantly more expressive than traditional (i.e., orthogonal) decision trees.

N-ary Differential Trees. In our experiments, we have only considered differential binary trees as this is the default setting of FFF [8]. Yet, it may be interesting to extend this approach to N-ary trees, to observe the complexity, memory, energy, and accuracy trade-offs. We believe this may potentially improve the accuracy, although at the cost of increasing the memory footprint, but this can be especially beneficial for more complex tasks.

Missing convolutional layers. One limitation of our current approach is that, while it performs effectively on time series data (such as HAR) or properly pre-processed speech data (such as KWS), when it comes to handling image classification its performances can be limited by the lack of convolutional layers, which as known allow for the possibility of recognizing and composing simple, local patterns (obviously, this also holds true for the other non-convolutional models, such as FCNs and FFF models). This ability is crucial to achieving state-of-the-art performance in complex computer vision tasks. While the results of a simple vision task as MNIST are encouraging, further investigation is needed to understand how to unroll (and prune) the convolutional operations in an effective way into the Fast-Inf architecture, e.g., based on multipartite graph

representations [20]. Future work should also investigate the use of vision Transformers based on Fast-Inf, similarly to what has been proposed in [8].

Input Size Overhead. One of the main sources of the computational overhead of Fast-Inf models is the input size. The larger the input, the more computations must be performed. We plan to explore solutions such as pre-computing high-level input features, inserting pooling layers, using convolutional filters (as done e.g. in [22]), or using dimensionality reduction techniques such as PCA.

Hardware Acceleration. Fast-Inf is a software-based and portable approach, but it can be further enhanced by incorporating hardware acceleration to improve inference efficiency. The MSP430FR series MCUs, equipped with the Low Energy Accelerator (LEA) [32, 44, 53], offer energy-efficient vector-based signal processing. LEA can be leveraged to offload MAC operations, especially in the leaf nodes of Fast-Inf models, to exploit parallelism and energy efficiency. However, LEA loses its computational state when there is a power failure, which requires repeated hardware reconfiguration and data transfer between volatile and non-volatile memory. Moreover, the sparsity introduced by the compression provides significant benefits for the software implementation, however, it introduces significant challenges for hardware acceleration [70]. We plan to explore hardware acceleration of Fast-Inf models in the future.

Other Resource-Efficient Inference Approaches. Various approaches, such as spiking neural networks [12] and Tsetlin machines [6], offer intelligence with lower resource requirements. However, it remains unclear which solution is superior in terms of accuracy, memory footprint, and energy efficiency, as there is no clear consensus in the literature. To make a fair comparison with such models, we contacted the authors of [6] to obtain the models used in their paper; unfortunately, they could not open-source their code. Furthermore, they did preprocessing on the open-source datasets they mentioned in their paper, but the modified datasets are also not open-source. Therefore, it is difficult to present a head-to-head comparison with those options in the literature. Still, Fast-Inf is superior concerning the reported evaluation in [6, Table 5] since they reported inference times on the order of **seconds**, while Fast-Inf completes inference on the order of **milliseconds**. To demonstrate the superiority of Fast-Inf over these approaches, it is necessary to conduct extensive research that employs different datasets, applications, and hardware platforms. We leave this as a topic for future exploration.

Other Compression Approaches. In Fast-Inf models, there is a trade-off between accuracy and memory consumption (as they are both related to the depth of the tree). Future work should aim to reduce the memory footprint of deep Fast-Inf models, to allow for larger depths and better accuracy on the extreme edge. Finally, future work should also focus on improving leaf utilization in Fast-Inf models.

7 Conclusions

In this paper, we introduced Fast-Inf, a new embedded intelligence solution for extremely resource-constrained batteryless edge devices. Fast-Inf uses binary tree-based neural networks that are ultra-fast and energy-efficient due to their logarithmic time complexity. We introduced adaptation and compression techniques to make Fast-Inf models energy-responsive and memory-efficient. We demonstrated that Fast-Inf models are significantly superior than existing approaches for batteryless systems, as they introduce much lower backup and runtime memory overhead during intermittent execution. We believe that Fast-Inf serves as a robust baseline for future research and provides a wide design space for further exploration and enhancements.

Acknowledgments

Funded by the European Union (project no. 101071179). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EISMEA. Neither the European Union nor the granting authority can be held responsible for them. We thank the anonymous reviewers of MobiSys 2024, MobiCom 2024, and SenSys 2024 for their valuable comments and feedback. We are also grateful to the SenSys shepherd for shepherding our final draft.

References

[1] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *Proceedings of the 17th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IEEE, New York, NY, USA, 188–199.

[2] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, et al. 2020. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proceedings of the 18th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 368–381.

[3] Saad Ahmed, Bashima Islam, Kasim Sinan Yildirim, Marco Zimmerling, Przemysław Pawełczak, Muhammad Hamad Alizai, Brandon Lucia, Luca Mottola, Jacob Sorber, and Josiah Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* 67, 3 (2024), 64–73.

[4] Khakim Akhunov and Kasım Sinan Yıldırım. 2022. AdaMICA: Adaptive Multicore Intermittent Computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 3 (2022), 1–30.

[5] Abu Bakar, Rishabh Goel, Jasper de Winkel, Jason Huang, Saad Ahmed, Bashima Islam, Przemysław Pawełczak, Kasım Sinan Yıldırım, and Josiah Hester. 2022. Protean: An energy-efficient and heterogeneous platform for adaptive and hardware-accelerated battery-free computing. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 207–221.

[6] Abu Bakar, Tousif Rahman, Alessandro Montanari, Jie Lei, Rishad Shafik, and Fahim Kawsar. 2022. Logic-based intelligence for batteryless sensors. In *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*. ACM, New York, NY, USA, 22–28.

[7] Abu Bakar, Alexander G Ross, Kasım Sinan Yıldırım, and Josiah Hester. 2021. REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 5, 3 (2021), 1–42.

[8] Peter Belcak and Roger Wattenhofer. 2023. Fast Feedforward Networks. arXiv preprint arXiv:2308.14711.

[9] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. 2015. Conditional computation in neural networks for faster models. arXiv preprint arXiv:1511.06297.

[10] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 176–189.

[11] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IEEE, New York, NY, USA, 209–219.

[12] Sizhen Bian and Michele Magno. 2023. Evaluating Spiking Neural Network on Neuromorphic Platform For Human Activity Recognition. In *Proceedings of the 2023 ACM International Symposium on Wearable Computers*. ACM, New York, NY, USA, 82–86.

[13] Leo Breiman. 2017. *Classification and regression trees*. Routledge, London, UK.

[14] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. 2022. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Transactions on Design Automation of Electronic Systems* 27, 3 (2022), 1–50.

[15] Luca Caronti, Khakim Akhunov, Matteo Nardello, Kasım Sinan Yıldırım, and Davide Brunelli. 2023. Fine-grained hardware acceleration for efficient batteryless intermittent inference on the edge. *ACM Transactions on Embedded Computing Systems* 22, 5 (2023), 1–19.

[16] Gianmarco Cerutti, Renzo Andri, Lukas Cavigelli, Elisabetta Farella, Michele Magno, and Luca Benini. 2020. Sound event detection with binary neural networks on tightly power-constrained IoT devices. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. IEEE, New York, NY, USA, 19–24.

[17] Gianmarco Cerutti, Lukas Cavigelli, Renzo Andri, Michele Magno, Elisabetta Farella, and Luca Benini. 2022. Sub-mW keyword spotting on an MCU: Analog binary feature extraction and binary neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 5 (2022), 2002–2012.

[18] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-directed high-performance intermittent computation with power failure immunity. In *Proceedings of the IEEE 28th Real-Time and Embedded Technology and Applications Symposium*. IEEE, New York, NY, USA, 40–54.

[19] Alexei Colin and Brandon Lucia. 2016. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 514–530.

[20] Elia Cunegatti, Doina Bucur, and Giovanni Iacca. 2023. Peeking inside Sparse Neural Networks using Multi-Partite Graph Representations. arXiv preprint arXiv:2305.16886.

[21] Leonardo Lucio Custode, Pietro Farina, Eren Yıldız, Renan Beran Kılıç, Kasım Sinan Yıldırım, and Giovanni Iacca. 2024. Github Repo. https://github.com/DIOL-UniTN/Fast-Inf-FFF.

[22] Leonardo Lucio Custode and Giovanni Iacca. 2022. Interpretable pipelines with evolutionary optimized modules for reinforcement learning tasks with visual inputs. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, USA, 224–227.

[23] Jasper De Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-free game boy. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 3 (2020), 1–34.

[24] Li Deng. 2012. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[25] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. 2022. Camaroptera: A long-range image sensor with local inference for remote sensing applications. *ACM Transactions on Embedded Computing Systems* 21, 3 (2022), 1–25.

[26] Digilent. [n. d.]. Analog Discovery 2 (Legacy). https://digilent.com/reference/test-and-measurement/analog-discovery-2/start

[27] Ferhat Erata, Eren Yildiz, Arda Goknil, Kasım Sinan Yıldırım, Jakub Szefer, Ruzica Piskac, and Gokcin Sezgin. 2023. ETAP: Energy-aware timing analysis of intermittent programs. *ACM Transactions on Embedded Computing Systems* 22, 2 (2023), 1–31.

[28] Pietro Farina, Subrata Biswas, Eren Yıldız, Khakim Akhunov, Saad Ahmed, Bashima Islam, and Kasım Sinan Yıldırım. 2024. Memory-efficient Energy-adaptive Inference of Pre-Trained Models on Batteryless Embedded Systems. arXiv preprint arXiv:2405.10426.

[29] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. 2020. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*. PMLR, Vienna, Austria, 3259–3269.

[30] Nicholas Frosst and Geoffrey Hinton. 2017. Distilling a Neural Network Into a Soft Decision Tree. arXiv preprint arXiv:1711.09784.

[31] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2022. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*. Chapman and Hall/CRC, Boca Raton, FL, USA, 291–326.

[32] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 199–213.

[33] Arda Goknil and Kasım Sinan Yıldırım. 2022. Toward Sustainable IoT Applications: Unique Challenges for Programming the Batteryless Edge. *IEEE Software* 39, 5 (2022), 92–100.

[34] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-sized machine learning models to tiny IoT devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 79–95.

[35] 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.

[36] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems* 28 (2015), 9.

[37] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 1–13.

[38] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 1–13.

[39] Andrey Ignatov. 2017. Real-time human activity recognition from accelerometer data using Convolutional Neural Networks. https://github.com/aiff22/HAR

[40] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf

[41] Texas Instruments Inc. 2019. EnergyTrace User Guide. https://software-dl.ti.com/simplelink/esd/simplelink_cc13x2_26x2_sdk/4.10.00.78/exports/docs/ble5stack/ble_user_guide/html/energy-trace/energy-trace.html

[42] Texas Instruments Inc. 2020. FRAM FAQs. https://www.ti.com/lit/wp/slat151/slat151.pdf

[43] Infineon. 2020. 8MB EXCELON LP Ferroelectric RAM. https://www.infineon.com/dgdl/Infineon-CY15B108QN_CY15V108QN_Excelon(TM)_LP_8-Mbit_(1024K_X_8)_Serial_(SPI)_F-RAM-DataSheet-v10_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0ee7134b6ff4

[44] Texas Instruments. 2016. Low-Energy Accelerator (LEA) Frequently Asked Questions (FAQ). https://www.ti.com/lit/an/slaa720/slaa720.pdf

[45] Bashima Islam and Shahriar Nirjon. 2020. Zygarde: Time-Sensitive On-Device Deep Inference and Adaptation on Intermittently-Powered Systems. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 3 (2020), 1–29.

[46] Seunghyeok Jeon, Yonghun Choi, Yeonwoo Cho, and Hojung Cha. 2023. HarvNet: Resource-Optimized Operation of Multi-Exit Deep Neural Networks on Energy Harvesting Devices. In *Proceedings of the 21st International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 42–55.

[47] Jeff Johnson. 2018. Rethinking floating point for deep learning. arXiv preprint arXiv:1811.01721.

[48] Kyle Johnson, Zachary Englhardt, Vicente Arroyos, Dennis Yin, Shwetak Patel, and Vikram Iyer. 2023. MilliMobile: An Autonomous Battery-free Wireless Microrobot. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. ACM, New York, NY, USA, 1–16.

[49] Chih-Kai Kang, Hashan Roshantha Mendis, Chun-Han Lin, Ming-Syan Chen, and Pi-Cheng Hsiu. 2020. Everything leaves footprints: Hardware accelerated intermittent deep inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3479–3491.

[50] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. arXiv preprint arXiv:1511.06530.

[51] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

[52] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 85–99.

[53] Seulki Lee and Shahriar Nirjon. 2019. Neuro.ZERO: a zero-energy neural network accelerator for embedded sensing and inference systems. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 138–152.

[54] Seulki Lee and Shahriar Nirjon. 2020. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 175–190.

[55] Seulki Lee and Shahriar Nirjon. 2022. Weight Separation for Memory-Efficient and Accurate Deep Multitask Learning. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*. IEEE, New York, NY, USA, 13–22.

[56] 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.

[57] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. arXiv preprint arXiv:1810.05270.

[58] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[59] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasım Sinan Yıldırım, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks* 16, 1 (2020), 1–24.

[60] Mahathir Monjur, Yubo Luo, Zhenyu Wang, and Shahriar Nirjon. 2023. SoundSieve: Seconds-Long Audio Event Recognition on Intermittently-Powered Systems. In *Proceedings of the 21st International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 28–41.

[61] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. 2020. ePerceptive: energy reactive embedded intelligence for batteryless sensors. In *Proceedings of the 18th ACM Conference on*

Embedded Networked Sensor Systems. ACM, New York, NY, USA, 382–394.

[62] Tushar S Muratkar, Ankit Bhurane, and Ashwin Kothari. 2020. Battery-less internet of things–A survey. *Computer Networks* 180 (2020), 107385.

[63] Powercast. 2016. The Powercast P2110B Powerharvester. https://www.powercastco.com/wp-content/uploads/2016/12/P2110B-Datasheet-Rev-3.pdf

[64] Powercast. 2018. The Powercast TX91501B Powercaster. https://www.powercastco.com/wp-content/uploads/2019/10/User-Manual-TX-915-01B-Rev-A-1.pdf

[65] Shvetank Prakash, Matthew Stewart, Colby Banbury, Mark Mazumder, Pete Warden, Brian Plancher, and Vijay Janapa Reddi. 2023. Is TinyML Sustainable? *Commun. ACM* 66, 11 (2023), 68–77.

[66] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary neural networks: A survey. *Pattern Recognition* 105 (2020), 107281. https://doi.org/10.1016/j.patcog.2020.107281

[67] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*. ACM, New York, NY, USA, 1–3.

[68] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, Cham, Switzerland, 525–542.

[69] Teodora Sanislav, George Dan Mois, Sherali Zeadally, and Silviu Corneliu Folea. 2021. Energy harvesting techniques for internet of things (IoT). *IEEE Access* 9 (2021), 39530–39549.

[70] Nathan Serafin, Souradip Ghosh, Harsh Desai, Nathan Beckmann, and Brandon Lucia. 2023. Pipestitch: An energy-minimal dataflow architecture with lightweight threads. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, New York, NY, USA, 14.

[71] Bharath Sudharsan, Sonu Prasad, Dan Jose, and John G Breslin. 2022. TMM-TinyML: tensor memory mapping (TMM) method for tiny machine learning (TinyML). In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. ACM, New York, NY, USA, 865–867.

[72] Bharath Sudharsan, Simone Salerno, and Rajiv Ranjan. 2022. TinyML-CAM: 80 FPS image recognition in 1 kB RAM. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. ACM, New York, NY, USA, 862–864.

[73] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. 2015. Convolutional neural networks with low-rank regularization. arXiv preprint arXiv:1511.06067.

[74] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. arXiv preprint arXiv:1804.03209.

[75] Pete Warden and Daniel Situnayake. 2019. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, Sebastopol, CA, USA.

[76] Harrison Williams and Matthew Hicks. 2024. Energy-adaptive Buffering for Efficient, Responsive, and Persistent Batteryless Systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 268–282.

[77] Fan Yang, Ashok Samraj Thangarajan, Sam Michiels, Wouter Joosen, and Danny Hughes. 2021. Morphy: Software defined charge storage for the iot. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. 248–260.

[78] Li Yang and Abdallah Shami. 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing* 415 (2020), 295–316. https://doi.org/10.1016/j.neucom.2020.07.061

[79] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 41–53.

[80] Eren Yildiz, Saad Ahmed, Bashima Islam, Josiah Hester, and Kasım Sinan Yıldırım. 2023. Efficient and Safe I/O Operations for Intermittent Systems. In *Proceedings of the 18th European Conference on Computer Systems*. ACM, New York, NY, USA, 63–78.

[81] Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. 2022. Immortal Threads: Multithreaded Event-driven Intermittent Computing on {Ultra-Low-Power} Microcontrollers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 339–355.

[82] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.