SUSTAIN Deliverable

D3.1 Data-driven model library for sensor networks

Grant Agreement number	101071179
Action Acronym	SUSTAIN
Action Title	Smart Building Sensitive to Daily Sentiment
Type of action:	HORIZON EIC Grants
Version date of the Annex I against which the assessment will be made	28th March 2022
Start date of the project	1st October 2022
Due date of the deliverable	M18
Actual date of submission	31st March 2024
Lead beneficiary for the deliverable	UNITN
Dissemination level of the deliverable	Public

Action coordinator's scientific representative

Prof. Stephan Sigg
AALTO –KORKEAKOULUSÄÄTIÖ,
Aalto University School of Electrical Engineering, Department of Communications and Networking
stephan.sigg@aalto.fi

Funded by the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Innovation Council and SMEs Executive Agency (EISMEA). Neither the European Union nor the granting authority can be held responsible for them.

European Innovation Council

Authors in alphabetical order		
Name	**Beneficiary**	**e-mail**
Leonardo Lucio Custode	UNITN	leonardo.custode@unitn.it
Giovanni Iacca	UNITN	giovanni.iacca@unitn.it
Kasim Sinan Yildirim	UNITN	kasimsinan.yildirim@unitn.it
Mir Hassan	UNITN	mir.hassan@unitn.it
Renan Beran Kilic	UNITN	renanberan.kilic@unitn.it

Abstract

This document presents the SUSTAIN **Deliverable D3.1 Data-driven model library for sensor networks**. This Deliverable is tightly coupled with **Task T3.1 Lightweight black-box and transparent models**, running from M1 to M18, and presents its main outcomes. The goal of this task was to design, implement and test data-driven models for sensor networks. As detailed in the present document, we have considered combinations of shallow neural networks and tree-based structures, properly designed to take into account computational (limited memory/CPU) & energy constraints. As we will see in this document, we have investigated the trade-offs between training time, amount of data needed for training, and model performance.

This Deliverable is meant to be public. It links to **D2.1 System modelling of the distributed and node intelligence: initial common strategy**, released in the early stages of the project, and to the forthcoming deliverables: **D3.2 Distributed learning framework** (to be released at M33), **D3.3 Probabilistic communication & computation system** (M36), and **D3.4 Comparative analysis of results** (M42).

This Deliverable is intended to be the final version of this document.

The code associated to this deliverable is publicly released at the following link: https://github.com/DIOL-UniTN/Fast-Inf

This deliverable is based on the following submissions (under submission at the time of writing this document):

1. Fast-Inf: Ultra-Fast Embedded Intelligence on the Batteryless Edge (Submitted to MobiCom 2024).
2. Memory-efficient Energy-adaptive Inference of Pre-Trained Models on Batteryless Embedded Systems (Submitted to EWSN 2024).

Table of Contents

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 [68]. 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 [56, 60, 61, 64]. 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 [53, 58]. Batteryless operation promises embedded intelligence forever without maintenance by using exclusively the energy harvested from the ambient [24]. 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 [6, 19, 39]. 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, 25]. Several recent studies have focused on batteryless intelligence and demonstrated the intermittent execution of tiny DNN models under resource constraints and stringent energy budgets [12, 24, 67]. However, 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 [24, 67]. Thus, several charge/compute cycles (power cycles) are required to complete the inference. At the end of each power cycle, the device backs up the computational state in nonvolatile memory. When it turns 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 [24, 36, 67], preventing timely responses.

P2: No response to charging time dynamics. Harvested energy can be often unstable due to the sporadic and uncontrollable nature of ambient energy sources. This can lead to longer charging times and increased latency during computations [20, 31, 43]. 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 [36, 37], 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.

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 [51]. 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 embedded intelligence solution specifically tailored to extremely resource-constrained systems. We build Fast-Inf around the recently proposed Fast Feedforward (FFF) architecture [8].

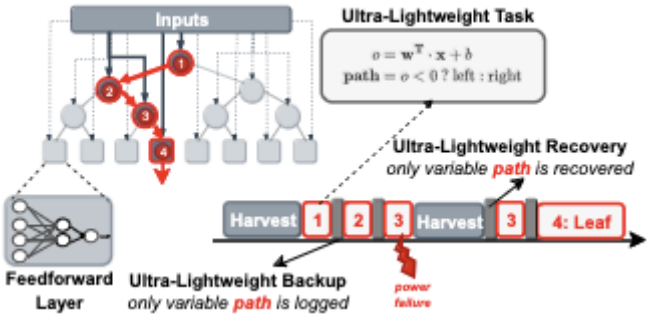

Figure 1: Conceptual scheme of Fast-Inf functioning.

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. When a leaf node is reached, the inference concludes by executing a feedforward layer. We have introduced several innovations on the original FFF networks, to ensure that Fast-Inf models can fit in the extreme edge:
(1) Importantly, FFF networks yield a memory/inference time and accuracy trade-off as deeper FFF 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 that makes these networks adaptive and energy-aware (i.e., aware of the energy harvesting dynamics).
(3) Besides, to improve the accuracy of original FFF networks, we have extended the training procedure introduced in [8] by using a different loss function.
(4) Finally, we introduced an energy- and memory-efficient inference engine that enables fast and efficient intermittent FFF inference with minimal overhead on the MSP430FR5994 [33] platform, which is the de facto standard 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 models. The Fast-Inf compression reduces the memory footprint of original FFF networks significantly (by up to 24.5x), making them fit in 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 608× speedup and less energy consumption compared to DNNs during our experiments on the MSP430FR5994 MCU.
(3) Tiny runtime. Fast-Inf inference engine has 6× less code size and 5600× smaller runtime buffer compared to the de facto inference engine for batteryless devices [24]. Fast-Inf inference tasks are lightweight and have significantly lower backup and runtime memory overhead.
(4) Adaptable latency. Fast-Inf can truncate (i.e., skip) the leaves when necessary, minimizing latency without degrading the accuracy significantly, and adapting inference to sporadic energy conditions and dynamic constraints. This strategy further reduces inference time by 6x.

We believe that Fast-Inf acts as a solid baseline for future research and leaves a rich design space for future works for further exploration and improvements. We make our code publicly available at [5].

2 Related Work

Enabling embedded intelligence on the batteryless edge offers intelligence by using the “free” energy from the environment [24, 25], but comes with several challenges due to extreme resource constraints and intermittent operation. In this section, we summarize these challenges, underline how Fast-Inf addresses them, and highlight the main differences between Fast-Inf and the state-of-the-art solutions.

2.1 Addressing Resource Constraints

Various techniques have been proposed to reduce the parameters of DNN models to make them suitable for embedded devices with limited memory and computational resources [9, 11, 28, 41]. These techniques include quantization [23, 26, 38], pruning [28, 29, 45, 46, 48], and separation [9, 24, 62], and have been already implemented in multiple studies targeting batteryless edge devices. For instance, Gobieski et al. [24] utilized pruning, separation, and neural architecture search [47] to obtain a DNN model that can meet the memory and energy requirements of the target device. However, the intermittent execution of these models 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 [6, 17, 19, 30]. 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 [57]. Several software solutions have been proposed to address these issues [7, 10, 13, 14, 43, 49, 50, 65, 67]. 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. Here, we consider the task-based model [7, 14, 49, 50, 65], 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 [14, 49, 50, 65, 66]. Several recent works focused on the task-based implementation of custom DNN workloads and their efficient intermittent execution [12, 24].

Computationally heavy tasks. Gobieski et al. [24] tested a task-based implementation of a DNN for the MNIST dataset [18]. This implementation involves 18 complex computational tasks, with the convolution task alone requiring several hundred thousand multiply-and-accumulate (MAC) operations. 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 [24, 67]. 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 [12, 24, 40, 44].

Backup and runtime memory overhead. MAC operations (e.g., o+=w*x+b) executed by DNN tasks have WAR dependency. To preserve idempotency and enable failure-atomic execution of these DNN tasks, their inputs

and outputs are separately maintained in non-volatile memory in a working buffer [24]. Consequently, the working 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 requirement. Unfortunately, DNNs introduce large runtime memory overhead due to several large layers being used. Besides, DNN tasks also need to back up their outputs upon completion. This is to avoid losing their computational results due to a power failure, but introducing significant energy and time overhead. In general, tasks with larger runtime buffer requirements have considerably larger backup overhead.

Latency Adaptation Overhead. Existing works used early exit branches to adapt inference time considering the available energy [36, 37, 52]. 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. Fast-Inf is significantly different from all prior works since it builds upon the recently introduced FFF networks [8], introducing several essential improvements to the original FFF architecture, which allows making it deployable on edge systems with extremely limited resources. In essence, Fast-Inf is the first pure software-based work that achieves ultra-fast inference, introduces minimal memory overhead, and offers adaptive latency. We identify three main features that make Fast-Inf distinct from the current state of the art.

1 Fast-Inf employs a custom loss function during training, which encourages a sort of "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 in the small memory of a batteryless edge.

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

3 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.

Table 1: Prior works on batteryless intelligence

Solutions	Features
Rehash [7], AdaMICA [4], Camaroptera [19], ImmortalThreads [67], Neuro.ZERO [44], Protean [6], SONIC [24], SoundSieve [51]	Compressed DNN models, slow inference, high energy overhead, no adaptable latency.
HarvNet [37], Zygarde [36], ePerceptive [52]	Compressed DNN models, slow inference, high energy overhead, adaptable latency via early-exit branches.
FAST-INF (this work)	**FFF** networks, **ultra-fast** inference, **very-low** energy overhead, **truncated** inference.

3 Fast-Inf on the Extreme Edge

Fast-Inf is a novel embedded intelligence solution specifically tailored to extremely resource-constrained batteryless systems. At the core of Fast-Inf is the tree-based FFF network architecture proposed in [8], which enables ultra-fast and energy-efficient inference due to its inherent logarithmic time complexity.

3.1 Overview of Fast-Inf

An FFF network is a neural network with a k-d tree structure. Fast-Inf employs FFF networks with a binary tree structure, depicted in Figure 1. Each inner node of the tree can be seen as a single neuron, while the leaves are small feedforward layers with a single hidden layer.

3.1.1 Performing Fast Inference. Formally, each inner node i of the tree computes an intermediate output oi 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 $y\hat{} = W_{l,h}{}^T x + b_{l,h}$.

3.1.2 Applicability. As we show through our experiments in Section 5, Fast-Inf is suitable for solving problems that can be addressed using fully connected networks (FCNs). This is because the original architecture of FFF networks [8] 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 [32]), 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) [63]. This is another common embedded application that performs speech recognition to identify a set of target keywords through a microphone.

3.2 Training Fast-Inf Models

The aim of the Fast-Inf training process is 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. As we discuss in Section 3.2.2, we modify the training procedure from [8] to improve the inference accuracy and also to facilitate model compression through pruning, reducing the memory footprint of the model as well as the inference time.

3.2.1 Background on Training FFF Networks. In this subsection, we summarize the training procedure in [8], which works as follows. During training, the output of the model is a weighted sum of the outputs of each leaf, as initially introduced in [22]. Practically, the outputs of the inner nodes are converted to probabilities (of taking the right branch):

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

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

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

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

$$\hat{y} = P_0\hat{y}_0 + P_1\hat{y}_1 + \ldots + P_N\hat{y}_N \quad (3)$$

where N is the number of leaves. At each training step, $y\hat{}$ is used to compute the loss for the backpropagation as:

$$\mathcal{L}_{FFF} = \mathcal{L}_{xh} + h \cdot \mathcal{L}_h \quad (4)$$

where $L_{x\hat{h}}$ is the cross-entropy loss between the logits $y\hat{}$ and the real classes y and

$$\mathcal{L}_h = \sum_{x \in X} \sum_{N \in \mathcal{N}} H(N(x)) \quad (5)$$

is the entropy of the nodes' output probabilities (the so-called "hardening loss" in [8], where *X* is the dataset, *N* is the set of nodes in the tree, and H is the entropy measure). After training, we "discretize" the tree by allowing it to perform inference in logarithmic time w.r.t. the total number of neurons in the leaves. We do so by simply taking the path with the highest likelihood, i.e.:

$$\mathbb{P}(right|i) = \begin{cases} 0 & o_i < 0 \\ 1 & \text{otherwise.} \end{cases} \quad (6)$$

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

3.2.2 Fast-Inf Training. The training procedure for Fast-Inf is different from the training approach in [8], as we: (1) do not use the hardening loss; and (2) employ an L_2 loss on the leaves' parameters.

We do not employ the hardening loss described in [8] as, from preliminary experiments, it showed poor test accuracy with small models, which are our target. This may be due to the fact that the hardening loss makes the optimization process harder and thus requires significantly more epochs to converge to a satisfactory accuracy. Moreover, we add an L_2 loss on the leaves' parameters, to: (1) make pruning easier (as we describe soon); (2) allow the leaves to have "simple" input-output models. The latter point here means that, since we have a simple model, each leaf tries to predict a small subset of classes, which allows us to perform leaf truncation (see Section 3.3). Thus, the loss used in our method, which we optimize using Adam [42], is the following:

$$\mathcal{L} = \mathcal{L}_{xh} + w_{L2} \cdot \mathcal{L}_2. \quad (7)$$

The first term, L_{xh}, is the cross-entropy loss between the targets and the model's outputs, as in Equation (4). The second term, instead, is the L_2 norm of the parameters used in the leaves. The L_2 loss on the leaves' parameters pushes unnecessary weights to 0, yielding several advantages: (1) it facilitates compression by means of pruning considering the magnitude of the weights [21, 24], 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.

3.3 Boosting Inference Efficiency

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

• the inference cost as θ ($d \cdot s_i + s_i \cdot w + w \cdot s_o$);

• the memory footprint as θ ($2^{d-1} \cdot s_i + 2^d \cdot s_i \cdot w + 2^d \cdot w \cdot s_o$).

Intuitively, deeper networks lead to an increased memory footprint and inference cost. 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).

3.3.1. Leaf Truncation. In an FFF network, 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 vital 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:

$$\max_i(\mathbb{P}(i|l)) > \xi \quad (8)$$

then we can set:

$$\hat{y}_l^* = \{\mathbb{P}(0|l), \ldots, \mathbb{P}(n|l)\} \quad (9)$$

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 L_2 loss during training (we will further demonstrate this experimentally in Figure 3). 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 the 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 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.

3.3.2 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, it is not always 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 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 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 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 *S*% of the weights with smaller magnitudes to 0. These steps can be formalized as:

$$\mathcal{W}_l \leftarrow \mathcal{W}_l - \eta \sum_i \nabla_{\mathcal{W}_l} \mathcal{L}_i \qquad (10)$$

$$\mathcal{W}_l \leftarrow \text{hardThreshold}(\mathcal{S}, \mathcal{W}_l). \qquad (11)$$

Note that the weights set to 0 at this point might reappear in the next epoch. Through multiple iterations, the weight matrix tends to stabilize (keeping only some specific nonzero weights), so that 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 [33] 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 [34] 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 Fast-Inf Training and Compression

To develop our Fast-Inf models, we started from the FFF implementation in PyTorch from [8]. We build on top of the fastfeedforward library, which implements FFFs networks. The Python code used for the training function is shown in Listing 1.

Listing 2, instead, shows the Python implementation of the leaf truncation mechanism.

```
def train(net, trainloader, epochs, norm_weight=0.0, lr=1e-2, weight_decay=0.0):
    """Train the network on the training set."""
    # Define loss and optimizer
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, weight_decay=weight_decay)

    # Train the network for the given number of epochs
    for _ in range(epochs):
        # Iterate over data
        for images, labels in trainloader:
            images, labels = images.to(DEVICE), labels.to(DEVICE)
            optimizer.zero_grad()
            loss = criterion(net(images), labels)
            l2loss = 0.0
            if hasattr(net, 'fff'):
                l2loss += norm_weight * net.fff.w1s.pow(2).sum()
                l2loss += norm_weight * net.fff.w2s.pow(2).sum()
            else:
                l2loss = 0.0
                if norm_weight != 0:
                    for x in net.parameters():
                        l2loss += x.pow(2).sum()
            loss += norm_weight * l2loss
            loss.backward()
            optimizer.step()
```

Listing 1: Code for training Fast-Inf.

```
def simplify_leaves(net, trainloader, outputs):
    y, leaves = (get_dist(net, trainloader))  # Retrieve the distribution of classes in the leaves
    y = y.cpu().detach().numpy()  # Move the class vector to CPU
    leaves = leaves.cpu().detach().numpy()  # Move the leaves' ids to CPU

    n_simplifications = 0
    ratios = {}
    for l in np.unique(leaves):
        ratios[l] = torch.zeros(outputs)
        indices = leaves == l

        for i in range(outputs):
            ratios[l][i] = (np.sum(y[indices] == i) / np.sum(indices))

        argmax = np.argmax(ratios[l])
        if ratios[l][argmax] > 0.7:
            output = torch.zeros(outputs)
            output[argmax] = 1
            self._fastinference[l] = output
            n_simplifications += 1
            print(f"Leaf {l} has been replaced with {argmax}")
    print(self._fastinference)
    return n_simplifications
```

Listing 2: Code for the leaf simplification mechanism.

See Section 5.1 for the hyperparameter values used in our implementation. After training, to perform leaf truncation, we execute a script that reloads the dataset and checks the a priori probabilities. After truncation, 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 as described in Section 3.3.2. We start the compression process by setting S = 0.45, i.e., with a sparsity level of 45%, and gradually increase it to reach the required size. We monitor the accuracy during retraining and reduce sparsity if the accuracy drop exceeds 3-5%, to prevent over-pruning. To address overfitting during retraining, we use an L_2 regularization term.

4.2 Fast-Inf Model Representation

After training 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 [28], 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 3.3.1.

4.3 Fast-Inf Intermittent Inference Engine

The Fast-Inf inference engine utilizes a task-based programming model [7, 14, 65] 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 Running Fast-Inf Model. 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 [67]. In order to prevent this issue, we incorporated loop continuation in our implementation, introduced by Gobieski et al. [24]. 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.

```
1  task_t leaf_t{
2   if(fast){ // if flag is set, then fast inference
3     result = fast_inference[node_id];
4     next_task(tree_t);
5   }else{
6     ... // run the feedforward layer
7   }
8  }
```

Listing 3: Pseudocode for 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 Listing 3, 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 [65]. 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 evaluated Fast-Inf by testing it on datasets from various fields. Firstly, we evaluated Fast-Inf via simulations to investigate the effect of the model parameterization and compression, whose results are reported in Section 5.1. Then, we evaluated the performance of Fast-Inf models in our testbed using MSP430FR5994 launchpad as target platform. We report the second part of the evaluation in Section 5.2.

Datasets. We used three different datasets representing different applications. We considered MNIST [18] as an instance of a simple, yet plausible image-based application. Secondly, we consider the Google Speech Commands v2 dataset with 10 classes for Keyword Spotting (KWS) [63], taken as an instance of an audio-based application. For KWS, we used 13 Mel-Frequency Cepstral Coefficients (MFCCs) after 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 [32], which classifies activities using accelerometer data and represents an example of a wearable application.

Model Structures. Table 2 presents the model structures used in our evaluations. As we present in Section 5.1.3, we compressed Fast-Inf models using the pruning approach described in Section 3.3.2 to fit them in the 256 kB FRAM of the MSP430FR5994 MCU. Similarly, we employed our unstructured pruning approach to compress and create sparse versions of the DNN models in this table.

Table 2: The structures of DNNs 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). In DNN models, batch normalization and max pooling layers are used after each C layer; dropout layers are added after F layers.

Dataset	MNIST		KWS [63]		HAR [32]	
Model	FAST-INF	DNN	FAST-INF	DNN	FAST-INF	DNN
Layers	D:3 L:4	C:4 F:5	D:4 L:4	C:4 F:3	D:3 L:16	C:6 F:5
No. parameters	$31k$	$444k$	$335k$	$102k$	$42k$	$413k$

5.1 Evaluation via Simulations

In this section, we will study the properties of Fast-Inf, comparing them with those of the baseline algorithm, i.e., FFF from [8]. During some of the comparisons, we will refer to the baseline approach as "Vanilla FFF", to remark that our approach is an extension of the baseline FFF algorithm, and thus it belongs to the same class of algorithms.

5.1.1 Evaluation Metrics. For the evaluation in simulation, the main evaluation metrics were: (1) the Test accuracy, and (2) the number of Floating-point Operations Per Second (FLOPs). For the former, we measured the accuracy of each model on the test set, obtained after the training and discretization processes. For the latter, we measured the average number of floating-point operations performed by each model, assuming that all the leaves have uniform probability.

5.1.2 Results. We compare the performance of Fast-Inf (uncompressed models) versus vanilla FFF models (trained using the original training approach described in [8]) in Figure 2. We performed training with the following parameters: depth ∈ {2, 3, 4}; leaf width ∈ {4, 8, 16, 32}; 10 epochs; and a w_{L2} of 0.01 for MNIST and HAR, 0.001 for KWS, as preliminary experiments showed that larger w_{L2} were hindering learning in KWS.

Note that, for each configuration, we performed 10 runs to assess the statistical repeatability of our method. From the figure, we can observe that the models trained using Fast-Inf often exhibit better performance than that of vanilla FFF models. The Wilcoxon rank-sum test (with $\alpha = 0.05$) confirms the superiority of our method in all scenarios (aggregating all accuracies for each dataset). Moreover, in most cases, the test accuracy achieved by Fast-Inf models seems to be less dependent on the leaf width than for FFF models. This is an extremely convenient feature, as it allows us to train models with small leaves, which use less memory and have faster inference time than models with larger leaves.

In Figure 3, we compare the impact on memory consumption and test accuracy of our proposed truncation mechanism. Here, we observe two interesting behaviors. Firstly, we note that, especially for models with larger leaves, Fast-Inf is able to significantly reduce memory consumption while vanilla FFF models cannot. Secondly, Fast-Inf does not suffer from large accuracy drops like FFF, confirming our intuition that the L_2 loss helps in "specializing" the leaves towards the prediction of single classes. Last but not least, not only in some cases do we observe that the test accuracy of the models after truncation is comparable to those using the full network, but we also observe that it is sometimes higher. This indicates that the leaf truncation mechanism may prevent overfitting.

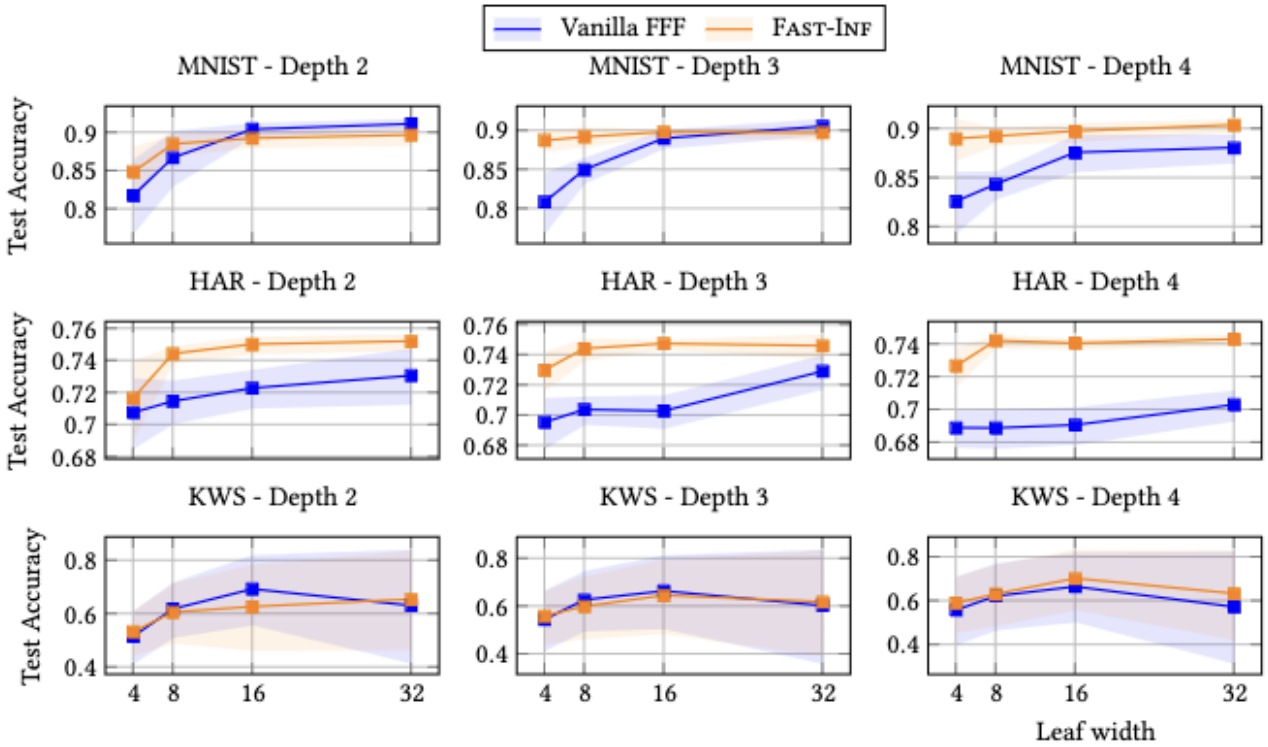

Figure 2: Test accuracy vs leaf width of vanilla FFF models vs Fast-Inf models.

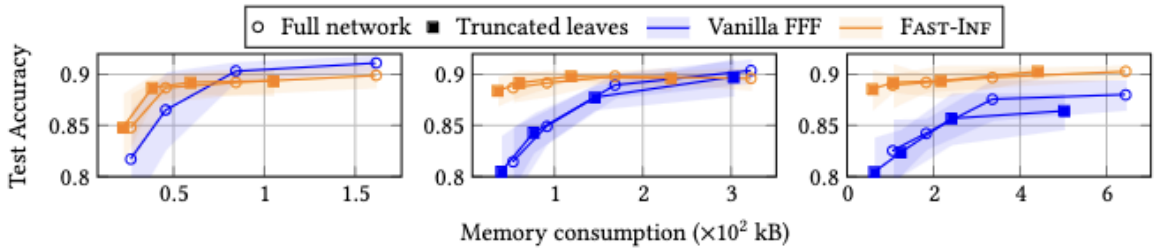

Figure 3: Test Accuracy vs memory consumption for various depths (left: 2, center: 3, right: 4), computed on the MNIST dataset. Note that, in the depth 2 scenario, we could not truncate any leaf from vanilla FFF models.

5.1.3 Compression of Fast-Inf models. Here, we will study the compression of Fast-Inf models, both from the point of view of leaf truncation and pruning.

Leaf Truncation Mechanism. In Figure 4, we show the impact of the L_2 loss on the number of leaves truncated on the trees. Note that, in our experiments, we fix $\xi = 0.7$ (unless stated otherwise). We observe that the introduction of the L2 term allows us to increase the number of leaves that can be truncated, confirming that our new loss allows us to perform faster inference. In Figure 5, we show the sensitivity of the models to the truncation threshold ξ. For the three values of depths, we observe that large leaves can be truncated at lower threshold values without any substantial loss in performance. This behavior may indicate that models that are too large tend to "overuse" some leaves and "underuse" others, which may negatively impact inference time. This also tells us that smaller models, besides using less memory, may be more efficient when using truncated inference. Moreover, we find that $\xi = 0.7$ can be a good threshold as it minimally impacts accuracy. However, at the extreme edges, we can avoid using the threshold and decide whether to perform truncated inference or query a leaf based on power constraints.

Unstructured Pruning. We also evaluate the performance of the depth-by-depth compression algorithm presented in Section 3.3.2, using as the main evaluation metric the accuracy over the model's size reduction. 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 on the nodes with the same depth and larger sizes, and optimizes the model to boost the accuracy. In Figure 6, we show the accuracy at each compression iteration. It can be clearly seen how this approach maintains, during compression, stable accuracy values up to significantly reduced dimensions.

Combined Pruning and Truncation. To minimize energy consumption, memory usage, and inference time, we combined leaf truncation and pruning, to achieve maximum compression. These two mechanisms are complementary and allow for massive savings in terms of both FLOPs and memory, as shown by the "T+C" points in Figure 7. 5.1.4

Comparison with the state of the art. In Table 3, we compare our results with those obtained using the baseline FFF [8] approach and the convolutional DNNs shown in Table 2. We also train FCN models with knowledge distillation [27], to study the time-memory trade-off between Fast-Inf and a model compression approach. We use a large FCN as the teacher model and a small FCN as the student model. The hidden layer width of the student model is set equal to the leaf width of the FFF model for the same task. This ensures that the inference times of both FFF models and distilled FCNs are comparable. We observe that Fast-Inf achieves comparable or even better performance w.r.t. the baseline approach from [8].

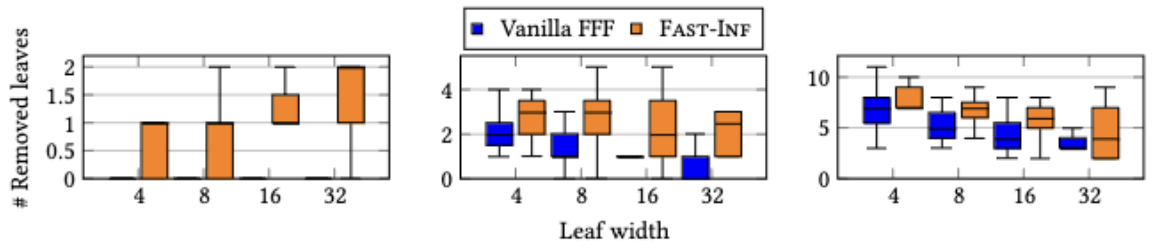

Figure 4: Impact of the L2 loss on the no. of leaves truncated at depth 2 (left), 3 (center), and 4 (right), computed on the MNIST dataset. In the experiments with depth 2, a significant portion of the trees could not be truncated due to the fact that the total number of leaves in the tree was small.

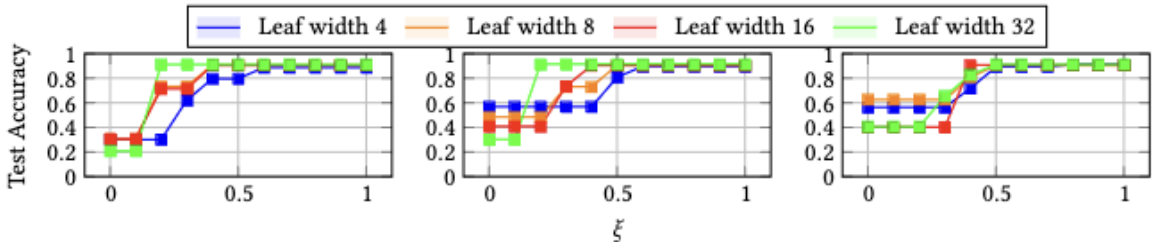

Figure 5: Test accuracy for various values of ξ at depth 2 (left), 3 (center), and 4 (right), computed on the MNIST dataset.

5.2 Evaluation on Real Hardware

We evaluated Fast-Inf by using the MSP430FR5994 launchpad. We ran the uncompressed Fast-Inf models (referred to as baseline models), and the compressed Fast-Inf and DNN models on this MCU intermittently and compared their performances considering several metrics. It is worth mentioning that baseline Fast-Inf models are, in general, more accurate than Fast-Inf models. Therefore, if baseline models already fit in memory, it is also desirable to execute them to obtain more accurate predictions. This is why we involved baseline HAR and MNIST Fast-Inf models in our evaluations.

Hardware Setup. We used the MSP430FR5994 MCU at 1 MHz. For repeatability, we emulated controlled power failures by utilizing a brown-out-reset mechanism of MCU every 5 ms to 20 ms. For the energy harvesting scenario, we used the Powercast TX91501-3W-ID power transmitter [55] and the P2110-EVB power harvester [54] equipped with a 1mF onboard energy storage supercapacitor.

Table 3: Comparison of CNNs, FCNs, FFF, and Fast-Inf. "(C)" stands for compression with an approximate size of 60 kB. "D" stands for distilled FCNs. "(∗)" indicates that, to use CNNs, we had to adopt a different pre-

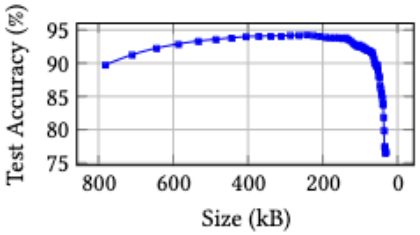

Figure 6: Gradual compression of a MNIST model:

processing technique (i.e., based on spectrograms).

Model	Test accuracy (%)		
	MNIST	HAR	KWS
CNNs	98	91	79(*)
CNNs (C)	99	92	82(*)
FCNs	97	85	86
FCNs (C)	97	86	78
FCNs (D)	96	81	72
FCNs (D, C)	94	80	72
FFF	92	76	82
FFF (C)	94	79	78
FAST-INF	92	76	83
FAST-INF (C)	97	80	77

accuracy is stable up to a significant reduction in size.

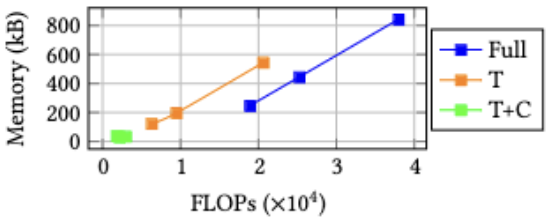

Figure 7: Memory size vs FLOPs for Fast-Inf models with depth 4 on the MNIST dataset. "Full" stands for the full Fast-Inf model, "T" stands for the model after truncation, and "T+C" stands for the model after truncation and compression.

Platform Specific Implementations. We implemented the compressed versions of the DNN model structures in Table 2 using Sonic [24], the de facto DNN-based intermittent inference engine for the extreme edge. The Sonic inference engine is based on the Alpaca [49] task-based model. We implemented Fast-Inf models using the Fast-Inf inference engine, which is also a task-based runtime. It is worth mentioning that Sonic optimized its models through a neural architecture search process, during which it explored different configurations by applying separation and pruning techniques to compress these networks and optimally balance their inference energy and accuracy considering the MSP430FR5994 platform. Using these optimal networks and the same target platform as in Sonic allowed us to present a systematic, rich, and fair comparison to state-of-the-art DNN models targeting battery-free systems.

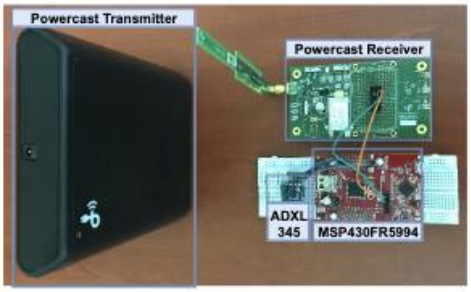

Figure 8: Real hardware evaluation setup.

Table 4: 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 (275x)	5	74	22	22
HAR	0.24	0.07 (257x)	5	56	18	22
KWS	0.18	0.06 (1020x)	5	123	61	22

5.2.1 Evaluation Metrics

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

5.2.2 Results

We present first our results collected during the continuously powered executions of our models as well as their intermittent execution under controlled power failures at uniform intervals.

5.2.3 *Inference Engine Overheads*

To understand how much backup/restore overhead is introduced by the Sonic and Fast-Inf inference engines, we implemented the "Pure C" versions of the DNN and Fast-Inf models that have no backup/restore overhead. We executed all models continuously without any power failures. We subtracted the execution time of Pure C versions from the execution time of the Fast-Inf and Sonic models to obtain the runtime

Table 5: Total execution time (sec) and energy consumption (mJ) during the continuous and intermittent execution.

Dataset	EXECUTION TIME (sec)										ENERGY CONSUMPTION (mJ)									
	FFF				Sonic		FCNs				FFF				Sonic		FCNs			
	Baseline		Fast-Inf		DNN		Dist.		Comp.		Baseline		Fast-Inf		DNN		Dist.		Comp.	
	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.	Cont.	Int.
MNIST	0.45 (208×)	0.47 (229×)	**0.4** (234×)	**0.42** (257×)	93.7	108.3	0.08	0.09	1.9	2.1	0.69 (208×)	0.91 (228×)	**0.62** (231×)	**0.76** (272×)	143.7	207.9	0.12	0.15	2.0	3.78
HAR	0.49 (152×)	0.51 (164×)	**0.31** (241×)	**0.33** (253×)	74.6	83.4	0.11	0.12	2.1	2.4	0.78 (174×)	0.97 (156×)	**0.59** (230×)	**0.71** (215×)	135.8	152.3	0.17	0.21	3.27	4.11
KWS	N/A	N/A	**0.32** (577×)	**0.36** (608×)	184.6	218.7	0.095	0.10	1.3	1.4	N/A	N/A	**0.50** (710×)	**0.78** (538×)	355.2	419.9	0.14	0.17	2.06	2.72

overhead introduced to run them intermittently, as presented in Table 4 under the "Runtime Ov." column. Thanks to ultra-efficient Fast-Inf models and their energy-efficient and lightweight characteristics, 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 small number of tasks required to be implemented to run the models, i.e., the Fast-Inf inference engine requires only 5 tasks, while the Sonic DNN engine requires 22 tasks to be implemented. Total Execution Time and Energy Consumption. We evaluated the execution time and energy consumption

Table 6: Detailed execution time profile of the nodes and leaves in Fast-Inf models.

Dataset	Nodes (ms)				Leaf (ms)			
	FFF	Fast-Inf			FFF	Fast-Inf		
		Min.	Avg.	Max.		Min.	Avg.	Max.
MNIST	61.0	55.8	70.3	85.5	266.6	68.2	148.0	234.9
HAR	25.1	27.0	29.4	31.9	412.7	141.9	222.1	285.5
KWS	N/A	3.0	29.6	87.2	N/A	9.3	89..4	299.0

Table 7: Memory overhead (in kB) of Fast-Inf and compressed base-line models. (Results for FCNs were similar to those of DNN and we omitted them due to space limitations)

	FFF			Fast-Inf			DNN		
	MNIST	HAR	KWS	MNIST	HAR	KWS	MNIST	HAR	KWS
.text	2.56	2.66	N/A	2.96	2.95	3.0	12.8	12.8	12.8
Data	220.8	94.3	N/A	42.4	40.3	40.2	153	30.4	219.4

of Fast-Inf and Sonic models during both continuous (Cont.) and intermittent (Int.) execution. Our results are presented in Table 5. We also included FCN models mentioned in Section 5.1.4 in this table. We observe that Fast-Inf models can reduce execution time by up to 608× compared to that of Sonic DNNs, respectively. These results are also aligned with the measured energy consumption. Fast-Inf can decrease energy consumption by up to 538× compared to Sonic DNNs. The execution time and energy consumption of Fast-Inf models are slightly higher compared to FCN models obtained via knowledge distillation, but the accuracy of Fast-Inf models is significantly better. Detailed Overheads of 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. Therefore, it takes a different amount of time for the Fast-Inf inference engine to perform sparse matrix multiplications and run each node and leaf. Table 6 shows the minimum, maximum, and average computation times for the leaves and nodes in Fast-Inf models. Note that nodes and leaves have a constant execution time in the baseline model since they receive the same inputs and perform the same amount of MAC operations. Besides, depending on the level of the sparsity, the compressed model representation might introduce a negligible

computational overhead for the inner nodes. Meanwhile, the computation time for the leaves is significantly reduced, e.g., up to 4× for MNIST.

5.3 Memory Footprint and Runtime Buffer Requirements.

We analyzed the .text and .data segments of the target binary to assess the memory footprint of the models and summarized our measurements in Table 7. Compared to Sonic inference engine, Fast-Inf inference engine, designed to operate with only 5 tasks, significantly reduces the code

Table 8: Runtime buffer requirements.

	MNIST	HAR	KWS
Fast-Inf	34 byte	50 byte	44 byte
Sonic	35 kB	9.6 kB	190 kB
FCN	2.3 kb	0.9 kb	2.4 kB

6 Conclusions and Discussion

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 FFF networks. 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 handle 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 the original FFF). This ability is crucial to achieving state-of-the-art performance in complex computer vision tasks. While the results on 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 FFF architecture, e.g., based on multipartite graph representations [15]. 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 [16]), or using dimensionality reduction techniques such as PCA.

6.1 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) [24, 35, 44], 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 [59]. We plan to explore hardware acceleration of Fast-Inf models in the future. 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.

1. 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] Fast-Inf code https://github.com/DIOL-UniTN/Fast-Inf.

[6] 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.

[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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. International Journal of Computer Vision 129, 6 (2021), 1789–1819.

[28] 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.

[29] 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.

[30] 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.

[31] 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, Anon. New York, NY, USA, 1–13.

[32] Andrey Ignatov. 2017. Real-time human activity recognition from accelerometer data using Convolutional Neural Networks. https: //github.com/aiff22/HAR [33] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf

[34] 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

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

[36] 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.

[37] 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.

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

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

[40] Chih-Kai Kang, Hashan Roshantha Mendis, Chun-Han Lin, MingSyan 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

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

[49] 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.

[50] 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 energyharvesting devices. ACM Transactions on Sensor Networks 16, 1 (2020), 1–24.

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

[52] 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.

[53] Tushar S Muratkar, Ankit Bhurane, and Ashwin Kothari. 2020. Batteryless internet of things–A survey. Computer Networks 180 (2020), 107385.

[54] Powercast. 2016. The Powercast P2110B Powerharvester. https: //www.powercastco.com/wp-content/uploads/2016/12/P2110BDatasheet-Rev-3.pdf The Powercast TX91501B Powercaster.

[55] Powercast. 2018. https://www.powercastco.com/wp-content/uploads/2019/10/UserManual-TX-915-01B-Rev-A-1.pdf

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Bharath Sudharsan, Simone Salerno, and Rajiv Ranjan. 2022. TinyMLCAM: 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.

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

[63] Pete Warden. 2018. Speech commands: A dataset for limitedvocabulary speech recognition. arXiv preprint arXiv:1804.03209.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

2. Appendix - Improving Fast-Inf performance on Cifar-10

Fast-Inf has an exceptional improvement in inference time. Yet, only being fast is never enough. In addition to that, we need to show that we can use Fast-Inf on real-world applications with sufficient performance. For that purpose, we explored an approach with autoencoders to improve the accuracy of Fast-Inf on Cifar-10, which is an image classification task.

Feedforward neural networks (FFs) are less complex than convolutional neural networks (CNNs). Yet, learning local features for FFs is more difficult than for CNNs. To overcome this issue, we proposed a pipeline using a convolutional autoencoder (CAE) for learning these local features and using only the encoder part of it during the inference.

Using the encoder of a pre-trained CAE has several advantages. In addition to their improvement on the performance, CAE also has a low number of parameters so that they do not introduce a significant memory footprint. CAE is also beneficial for on-device learning since it often does not require training with new real-time data. Yet, there is a trade-off. The complexity of the overall process increases. CNNs are computationally very expensive even though they don't have much memory footprint. Still, one can trade-off accuracy towards complexity. For that reason, we have introduced several CAEs with varying complexities to analyze the complexity-accuracy trade-off. Listing 4 shows several possible architectures and the code for generating the CAE after specifying the architecture. One can easily extend that code for implementing a neural architecture search (NAS) algorithm using CAE and Fast-Inf for finding an optimum architecture for the encoder with respect to accuracy.. We will investigate this approach in our later research.

```
from torch import nn

BASE_ARCHS = [
        [32, 64, 128],
        [16, 32, 64],
        [8, 16, 32],
        [64, 128],
        [32, 64],
        [16, 32],
        ]

class BaseAE(nn.Module):
+-- 12 lines: def __init__(self, in_chan, arch_num):

def build_ae(arch: list[int]):
    encoder_list, decoder_list = [], []
    depth = len(arch)
    for i in range(depth-1):
        last_layer = (i == depth - 2)
        en_ch_in, en_ch_out = arch[i], arch[i+1]
        dec_ch_in, dec_ch_out = arch[depth-i-1], arch[depth-i-2]
        encoder_list.append(DoubleConv(en_ch_in, en_ch_out))
        decoder_list.append(DoubleConv(dec_ch_in, dec_ch_out, last_layer))
        if i != depth - 2:
            encoder_list.append(nn.MaxPool2d(2))
            decoder_list.append(nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True))
    return nn.Sequential(*encoder_list), nn.Sequential(*decoder_list)
```

Listing 4: Code for generating the CAE based on the configuration

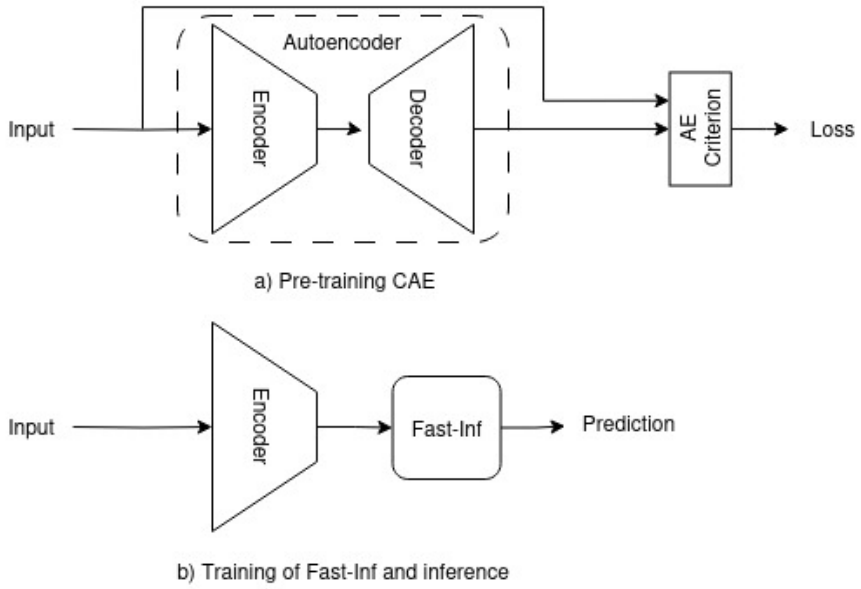

Figure 9: Pre-training CAE and using encoder side during training Fast-Inf and inference

Figure 9 shows the overall process. After pre-training CAE (Figure 9.a), we only use encoder-side during training and inference of Fast-Inf. Note that encoder is not re-trained during training training of Fast-Inf (Figure 9.b). As we mentioned, such an architecture is also beneficial for on-device learning since we only train it online so only the activations and gradients of Fast-Inf needs to fit inside a device's memory for backpropagation and not the ones from the encoder.

Table 9. Experiment Results

Model	Accuracy (%)	No. Parameters (Million)	MACs (Million)
Fast-Inf	38.4	0.84	0.05
Fast-Inf + Enc_1	62.8	1.06	0.41
Fast-Inf + Enc_2	65.1	1.31	0.84
Fast-Inf + Enc_3	66.4	1.87	1.51
Fast-Inf + Enc_4	66.8	2.53	3.86

Table 9 shows the performance measures of our experiments during inference. We used two complexity measures, the multiply-accumulate operations (MACs) and the number of parameters, both with the encoder and Fast-Inf. We observe that even the smallest encoder improves the accuracy significantly thanks to the fact that convolutional layers extract local features better than fully-connected layers. In addition, increasing the size of the encoder improves the performance as expected, yet, introducing significant complexity.