Memory-efficient Energy-adaptive Inference of Pre-Trained Models on Batteryless Embedded Systems

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

Subrata Biswas*
Worcester Polytechnic Institute
Worcester, USA
sbiswas@wpi.edu

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

Khakim Akhunov
University of Trento
Trento, Italy
khakim.akhunov@unitn.it

Saad Ahmed
Georgia Institute of Technology
Atlanta, USA
sahmed@gatech.edu

Bashima Islam
Worcester Polytechnic Institute
Worcester, USA
bislam@wpi.edu

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

Abstract

Batteryless systems frequently face power failures, requiring extra runtime buffers to maintain inference progress and leaving only a memory space for storing ultra-tiny deep neural networks (DNNs). Besides, making these models responsive to stochastic energy harvesting dynamics during inference requires a balance between inference accuracy, latency, and energy overhead. Recent works on compression mostly focus on time and memory, but often ignore energy dynamics or significantly reduce the accuracy of pre-trained DNNs. Existing energy-adaptive inference works modify the architecture of pre-trained models and have significant memory overhead. Thus, energy-adaptive and accurate inference of pre-trained DNNs on batteryless devices with extreme memory constraints is more challenging than traditional microcontrollers.

We combat these issues by proposing FreeML, a framework to optimize pre-trained DNN models for memory-efficient and energy-adaptive inference on batteryless systems. FreeML comprises (1) a novel compression technique to reduce the model footprint and runtime memory requirements simultaneously, making them executable on extremely memory-constrained batteryless platforms; and (2) the first early exit mechanism that uses a single exit branch for all exit points to terminate inference at any time, making models energy-adaptive with minimal memory overhead. Our experiments showed that FreeML reduces the model sizes by up to 95×, supports adaptive inference with a $2.03 - 19.65\times$ less memory overhead, and provides significant time and energy benefits with only a negligible accuracy drop compared to the state-of-the-art.

CCS Concepts

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

Keywords

Batteryless embedded systems, Deep Neural Networks (DNN), Model Compression, Global Early Exit

*Both authors contributed equally to the paper

1 Introduction

Advances in electronics and energy harvesting have given rise to batteryless devices that exclusively rely on ambient energy [6, 12]. These devices compute *intermittently* due to extremely scarce and transient ambient energy, creating significant challenges in hardware and software design [42]. Despite these challenges, the application space of intermittent computing is rapidly expanding [3].

On-device intelligence has become increasingly important for batteryless edge applications since it offers more efficient, reliable, timely, and secure computing solutions [14, 22]. Deep neural network (DNN) inference is feasible on batteryless sensing platforms, and it enhances the systems efficiency and throughput considerably [7, 22, 34]. However, deploying a pre-trained DNN model on an extremely resource-constrained batteryless device and running it intermittently pose significant problems to tackle.

P1: Memory Scarcity. Batteryless platforms are extremely memory-constrained, often having kilobyte-sized nonvolatile memory, e.g., 256KB FRAM [21], and only a few kilobytes of SRAM. FRAM is mainly used to back up and recover program data and states for power failure-resilient intermittent computation. Besides, inference requires extra memory for backup and recovery since input and output activations of the layers need to be preserved in FRAM [14]. Hence, only the remaining part of the FRAM (often less than half of its total size) can be used to store the parameters of the model. While traditional compression techniques can fit a pre-trained DNN model in 256KB by preserving its precision [9, 16, 29], they are insufficient when the memory available to store and execute the model is significantly smaller, as they result in a significant drop in accuracy [22]. On the other hand, existing works on batteryless systems obtain small models in a rigid and costly way: generate several compressed networks, retrain them, and perform an expensive search to find the best network [14]. Besides, they do not consider the energy dynamics and runtime memory requirements, which require support for different compression scales. Therefore, we need a solution to deploy pre-trained DNNs on batteryless systems smoothly by bridging the accuracy gap while matching the extreme memory and energy requirements.

P2: Energy Dynamics, Memory, and Latency. Unstable and sporadic availability of harvestable ambient energy can prevent

executing inference intermittently with an acceptable latency. Existing works addressed this issue by adapting inference accuracy concerning the available energy by – (1) maintaining multiple versions of the same model that offer different accuracies and latencies [6, 7, 26], which is memory-inefficient for a batteryless device; or (2) constructing models with early exit branches [22, 34], enabling the early termination of inference and providing *anytime* output with reasonable accuracy. Early-exit introduces significant memory overhead to maintain parameters for each exit branch and needs to keep input activations of these networks to provide results at any time. Besides, all these approaches require either changing the model structure or retraining the whole model and thus do not support working with pre-trained DNN models. Therefore, a plug-and-play early exit solution that conforms to the extreme memory constraints by introducing minimal memory overhead without altering the baseline DNN model is required.

Contributions. To fill this gap, we introduce FreeML—a systematic pipeline to optimize pre-trained DNN models for memory-efficient and energy-adaptive inference on batteryless systems. FreeML comes with the following specific features:

*(1) Sparsity-imposed DNN Compression (**SparseComp**)* is a new iterative algorithm that compresses selective layers of a pre-trained DNN by retraining and imposing sparsity constraints simultaneously. SparseComp minimizes accuracy degradation due to compression by retraining *only* the selected layers using a *small percent of the training data*. SparseComp does not employ expensive solutions, such as neural architecture search [14, 33], that require retraining of the whole model and fine-grained search within a large configuration space. In addition, SparseComp employs layer separation to reduce the amount of runtime memory space needed to store layer activations during intermittent inference.

*(2) Global Early Exit Network (**gNet**)* is the first plug-and-play early exit architecture that uses a single exit branch to exit from any layer of the network, introducing minimal memory overhead and eliminating the need to retrain or restructure the model. It is a one-architecture-fits-all network, which inserts only a single exit branch into the pre-trained model instead of one branch for each layer. This single exit branch takes the output from *all intermediate layers* as input using a unique *pooling mechanism* that handles missing intermediate outputs from the later layers. Therefore, it is possible to terminate DNN inference at any time by providing inference results without buffering previous layer outputs. Finally, gNet is the first early-exit model that removes joint-training of exit branches and the DNN model, or alternating base DNN architecture, which makes it suitable for pre-trained networks.

Our experiments show that SparseComp can achieve even 95× compression rates on pre-trained DNN models, i.e., reducing their sizes from MBs into a few KBs without a significant accuracy drop. gNet reduces memory overhead $2.03x - 19.65x$ times by replacing multiple exit branches of traditional early-exit models with a single global branch. Moreover, gNet reduces the inference time by 10.84% – 16.19% with a median accuracy gain of 2% compared to the traditional early-exit models. Thanks to SparseComp and gNet, FreeML provides significant time and energy benefits during intermittent inference by terminating ultra-tiny DNN inference anytime and providing timely outputs.

We release FreeML as an open-source framework via [1] to facilitate the automatic deployment and intermittent execution of DNN models, similar to the popular end-to-end frameworks Tensorflow-Micro and EdgeImpulse [19], which do not support the microcontrollers [13, 21] commonly used in batteryless systems. Our pipeline automatically converts the adapted and compressed DNN model into a set of portable source files. These files are linked with our DNN and intermittency control libraries that can execute the model intermittently and adaptively on batteryless devices.

2 DNN Inference on Intermittent Power

Batteryless devices use energy harvesters to capture ambient energy and store it in their small capacitors. Due to capacitors' limited capacity and ambient energy variability, these devices experience frequent power failures. Therefore, they perform computes *intermittently* by saving their computational state when power failure is imminent and restoring it when sufficient energy becomes available for resumption [2, 8, 11, 32, 41, 43]. Several works have demonstrated the intermittent execution of *custom and manually-optimized* tiny DNN models in batteryless platforms [14, 22, 26, 33, 34]. However, instead of resorting to these handcrafted DNN models, we need a systematic approach to adapt existing pre-trained models to conform to the severe memory and energy limitations of batteryless systems and to execute them intermittently.

2.1 SOTA: Deploying Pre-trained DNN Models

Model Compression: Several studies proposed DNN compression techniques to reduce their memory footprint and computational overhead to make them run efficiently on *mobile systems* with MB-sized memory and power-hungry MCUs. These techniques include threshold-based pruning connections with weights [16, 17] or using Fisher Information to prune unimportant connections [27, 28, 30], sparsifying fully-connected layers and separating convolutional kernels with tensor decomposition and low-rank approximation [9, 24]. Some studies compress DNNs to reduce their execution time and energy consumption [39, 40]. Unfortunately, these techniques ignore the additional memory requirement for intermittent operation, resulting in considerable accuracy drops when obtaining ultra-tiny models deployable on batteryless platforms [10].

Several intermittent systems [22, 26, 34] have utilized similar compression techniques on shallower and relatively smaller pre-trained DNNs by employing lower compression ratios to prevent accuracy degradation. Gobieski et al. [14] proposed a neural architecture search (NAS)-based approach that creates several compressed configurations of a DNN (through separation and pruning) to select the most accurate and energy-efficient configuration to fit the target device. However, the NAS requires exhaustive retraining and fine-grained search within a large search space, which is time-consuming and computationally expensive. Additionally, the search space, search algorithm, and performance estimation strategy should be well-defined to obtain promising results. In summary, we need a customizable compression technique to deploy pre-trained DNNs on batteryless systems smoothly by bridging the accuracy gap while matching the extreme memory requirements.

Early Exit: Early-exit models [36] allow dynamic reduction of inference time without sacrificing performance by introducing multiple exit branches in the network. Later works [22, 23, 34] expands

Table 1: A comparison of prior works on DNN compression and adaptive execution in batteryless systems.

Prior Works	DNN Model Compression	DNN Model Adaptation
Rehash [8], AdaMICA [4], Camaroptera [12], ImmortalThreads [43], Neuro.ZERO [26], LiteTM [7], Protean [6]	No ✗	No ✗
HarvNet [23]	No ✗	Memory inefficient early exit branches, separate models for each branch
SONIC & TAILS [14]	Compressing handcrafted small models via post-facto pruning, separation for only reducing model parameters, NAS for searching the best tiny model	No ✗
Zygarde [22]	Compressing handcrafted small models via post-facto pruning and separation for reducing model parameters	Memory inefficient early exit branches, separate models for each branch
ePerceptive [34]	Compressing handcrafted small models via post-facto pruning and separation for reducing model parameters	Memory inefficient early exit branches, separate models for each branch
FreeML (this work)	Compressing *any pre-trained DNN models* via **sparsity imposed retraining** for reducing *model parameters* and layer separation for reducing model *runtime memory requirements* → **Ultra-tiny DNN models**✓	*Plug-and-play early exit branches* via a single **global exit layer** that supports anytime output without altering the baseline model → **Memory-efficient early exit**✓

early-exit to address the energy sporadicity of batteryless systems by trading off the accuracy, latency, and energy constraints and integrating energy as one of the deciding parameters for early-exit models. However, these works require storing multiple exit branches, increasing the memory overhead with the number of exits. Though Zygarde [22] reduces the overhead by using clustering as an exit branch instead of a neural network, this still adds memory requirement for n clusters and changes the network architecture to a Siamese network. HarvNet [23] uses NAS to find the optimum number of exit points, which still requires storing the parameters of multiple exit branches. Therefore, we need to consider memory as one of the constraints in designing these early-exit models.

Besides, having multiple exit branches also requires either finishing the execution of an exit branch or storing the entire output required to execute the previous exit branch in a buffer if any interruption occurs. This introduces additional computation and memory overhead to provide "any-time output." Thus, a solution is needed to support any-time output without storing the entire output in buffers or always executing the previous exit branches.

Moreover, all these works (both batteryless and non-batteryless) require either solving a joint optimization to retrain the exit branch and the baseline neural network [34, 36] or changing the baseline architecture [22] which is computationally expensive and requires access to training dataset is not always accessible. Such retraining hinders us from using pre-trained models that are gaining popularity. Therefore, a "plug and play" early exit mechanism is needed to introduce dynamic inference without altering the baseline DNN.

2.2 Unique Features of FreeML

Table 1 compares FreeML with previous studies that demonstrated intermittent DNN inference on batteryless systems. FreeML is the first that provides a systematic pipeline to generate energy-aware adaptive and ultra-compressed accurate models from pre-trained DNNs, facilitating the deployment and intermittent execution of these models on real batteryless hardware platforms. Prior works are limited to the efficient intermittent execution of a few *handcrafted and optimized* small DNNs. FreeML addresses two main concerns simultaneously: memory footprint and energy awareness.

FreeML introduces SparseComp, a new approach that casts pruning into a *constrained optimization problem*, which aims to maximize the model accuracy while meeting the memory constraints. This is accomplished by *automatically but selectively retraining* large layers using only a small subset of the training data while simultaneously *imposing sparsity*. This strategy avoids exhaustive NAS to find the optimal model that meets the memory footprint requirements. SparseComp is not a trivial algorithm since it (1) imposes a different sparsity constraint for each layer to achieve minimal accuracy degradation and (2) reduces the runtime memory requirements needed to execute the model on the device.

In traditional early exit models, multiple exit branches need to be stored. FreeML employs gNet, the *first one-architecture-fits-all early-exit network* where a single network branch works as the exit point for all or selected [23] layers in the network. It is a "plug and play" exit model that supports "any-time output" with negligible overhead by extracting and storing the significant components of all or selected layer outputs in a single input buffer. gNet does not require altering the baseline model and is the first work that supports inserting exit layers on any pre-trained network. However, developing and training gNet is non-trivial for three reasons: (1) different layers' outputs vary in shape, making a one-fit-all model harder to achieve; (2) the output of later layers is not present when exiting from a previous layer; and (3) the memory, time, and computation overhead of the exit branch needs to be minimal.

3 FreeML for DNN Intermittent Inference

The FreeML pipeline follows a two-phase workflow. In the first phase, a given pre-trained DNN model is compressed to an ultra-tiny model (*SparseComp*) to fit into the target device's memory. Then, a global early exit network is augmented to the ultra-tiny model (*gNet*) to create an energy-aware and adaptive model. The second phase involves converting the compressed adaptive DNN model into C code that is linked with the FreeML ML library, which facilitates power-failure-resilient inference on the target platform.

3.1 Sparsity-imposed Compression of Models

SparseComp proposes a unique approach by treating DNN model compression as a *constrained optimization* problem. The goal of this optimization is to reduce the model's memory footprint so that it can fit into the limited memory of the target device while minimizing the accuracy drop that occurs due to compression. To solve this optimization problem, SparseComp selectively *retrains* the large layers of the pre-trained DNN model while imposing *autonomously identified* sparsity constraints during training. In addition to reducing the number of parameters, SparseComp minimizes the model's

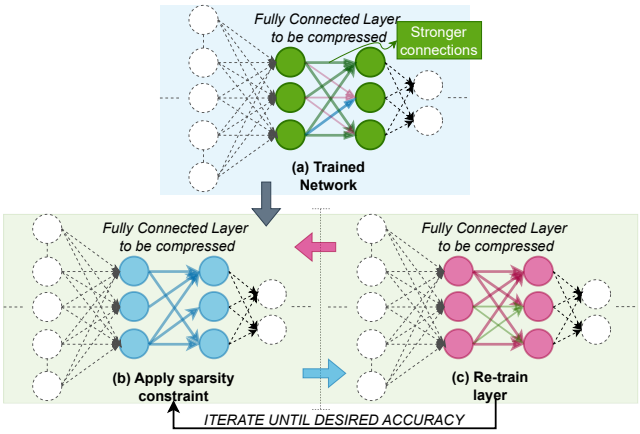

Figure 1: The SparseComp compression scheme. Smaller weights are pruned after imposing the sparsity constraint. The pruned weights can appear in the next epoch again during re-training. After several iterations, the layer is compressed with a minimal drop in the model accuracy.

runtime memory requirements during intermittent inference by reducing the memory needed to store intermediate results.

3.1.1 Overview. SparseComp includes two main components:

(1) Runtime-Aware Separation of Layers: To enable failure-atomic intermittent execution of DNN inference, FreeML maintains an internal *working buffer* in nonvolatile memory whose size is sufficient to keep the corresponding layer's both input and output activations. This is mandatory to make inference computations idempotent so that FreeML never reads and then writes to the same memory location, eliminating memory inconsistencies due to write-after-read (WAR) dependencies [43]. SparseComp applies **separation** techniques [9, 14] to split the layers that increase working buffer requirements, decreasing their runtime memory overhead with a minimal drop in model accuracy.

(2) Iterative Unstructured Pruning: After separating the layers of the pre-trained DNN model and reducing its runtime memory requirement, the next step is systematic pruning to compress and fit that model into the target device's memory. To achieve this, SparseComp follows a process where it selects the largest fully connected or convolutional layer in the model and defines a **sparsity constraint** for it. To compress the selected layer, we freeze all the preceding layers, i.e., their weights, biases, or filters are not modified, and only retrain the selected layer and the following ones by imposing the selected sparsity constraint **only** on the layer to be compressed. We impose the sparsity constraint using *projected gradient descent* that employs iterative hard-thresholding [15], allowing the memory constraints to always be respected on the selected layer during retraining. Once the selected large layer is retrained to achieve the desired sparsity with minimal degradation in the model accuracy, we start a new re-training iteration and repeat the process until the desired model size requirements are met.

3.1.2 Runtime-Aware Separation of Layers During deep neural network (DNN) inference at runtime, a common step of executing these layers is the multiply-and-accumulate (MAC) operation, which involves computing the product of two numbers and adding that product to an accumulator ($x+ = y * z$). However, performing MAC operations can cause anti-dependencies (i.e., WAR dependencies) [11, 14] since they need to read and write to the same memory locations in non-volatile memory, i.e., the accumulator x. If these operations are repeated due to power-failure interruptions, they can lead to different results due to anti-dependencies, making them inherently non-idempotent (i.e., not power-failure-resilient).

To address this issue, the FreeML library has a dedicated working buffer in non-volatile memory to maintain input and output activations of layers separately during inference at runtime. This separation ensures that FreeML never reads and then writes to the same memory locations, thereby enabling power-failure-resilient execution of each layer. The runtime memory requirement of a layer is the sum of its input and output activations. Therefore, the layer with the maximum runtime buffer size requirement specifies the size of the FreeML working buffer. FreeML employs the *separation* of layers to decrease their input/output sizes and, in turn, to reduce the working buffer size.

To reduce the working buffer requirements, SparseComp uses separation techniques to separate convolution layers with the Tucker tensor decomposition and fully connected layers with singular value decomposition [9, 14]. Previous approaches have mainly focused on reducing the number of multiplications and improving computational efficiency, but our approach also considers working buffer sizes. SparseComp starts with the layer that requires the largest buffer and separates it. For instance, a fully connected layer with dimensions $m \times n$ is factorized into two layers with dimensions $m \times k$ and $k \times n$ layers where $k < m$. In the prior case, the working buffer requirement for the layer is $m + n$, while in the latter, it is $m + k$ if we assume $m > n$. Separating layers reduces the working buffer size and energy requirements, but this comes at the cost of a drop in accuracy. We used Bayesian matrix factorization [35] to estimate and select the dimension of the inserted layer (i.e., k).

3.1.3 Iterative Unstructured Pruning After reducing the working buffer requirements of the model through the separation of the layers, SparseComp starts compressing the pre-trained DNN model layer by layer. Instead of pruning all layers non-selectively, SparseComp applies a specific compression rate to each layer and removes the risk of over-pruning the most meaningful layers. The compression of a layer is a two-step iterative process: First, a sparsity ratio is *automatically* selected for the layer to be compressed, and then the model accuracy is *optimized* by retraining that layer and subsequent layers in the model by *continuously imposing* the selected layer sparsity. These steps are repeated until the best sparsity ratio that does not significantly degrade model accuracy is selected to compress the layer. SparseComp then proceeds to the next layer that contributes the most to memory requirements and employs the same steps until the desired compression rate is achieved. Throughout the compression process, only one layer changes in size with each compression iteration. This approach allows SparseComp to assess the impact of compression on accuracy and adjust the compression rate for layers that exhibit significant accuracy drops.

(1) Automatic Selection of Sparsity. When choosing the sparsity value for compression, i.e., the number of zero-valued elements divided by the total number of elements, SparseComp considers the size of the layer to be compressed, the overall model size, and the target size of the compressed model. To prune larger layers, which

contribute more to the model size, SparseComp sets the initial sparsity to 0.9 (the percentage of non-zero model parameters), resulting in a 90% weight pruning. For smaller layers, SparseComp selects a higher initial sparsity value. After retraining the layer, SparseComp evaluates the model's accuracy. If the drop in accuracy is negligible, it increases the compression rate on the layer and repeats the process. When the accuracy drop exceeds a certain threshold (usually 3-5%), SparseComp reduces the compression rate for the next iteration by increasing the sparsity value and moves to the next largest layer. If the accuracy drops significantly, SparseComp defines the layer as fragile and excludes it from future compressions. As the desired compression ratio is approached, SparseComp selects less significant sparsity values to avoid over-pruning.

Algorithm 1: Compression of a Fully Connected Layer

Inputs: $\mathcal{M}$—pre-trained DNN model, $\mathcal{W}$—weight matrix of the FC layer, $\mathcal{S}$—sparsity value, $\mathcal{D}$— dataset of (X=value, Y=label) pairs.
Parameters: e—SGD epochs, B—batch size, η—learning rate.

1 **repeat**
2 pick B samples randomly from dataset D
3 $\hat{Y} = \texttt{forward}(\mathcal{M}, \mathcal{B})$ ▷ forward pass to get predictions
4 $\mathcal{W} \leftarrow \mathcal{W} - \eta \left(\sum_{i=0}^{B} \nabla_{\mathcal{W}} \mathcal{L}_i(y_i, \hat{y}_i)\right)$ ▷ compute gradient
5 $\mathcal{W} \leftarrow \texttt{hardThreshold}(\mathcal{W}, \mathcal{S})$ ▷ apply sparsity constraint
until *e epochs*

(2) Constrained Optimization with Re-training. The formulation of the problem is presented in eq. (1). Given input x and the set $\mathbf{W}$ of the layers' weights, the main objective is to minimize the empirical error $\mathcal{E}_{emp}$ of the model prediction $f(x, \mathbf{W})$ where $\|\mathcal{W}\|_0$ is the number of zero entries in $\mathcal{W} \in \mathbf{W}$ and $s_{\mathcal{W}}$ is the sparsity constraint associated to that layer.

$$\min_{\forall \mathcal{W} \in \mathbf{W}: \frac{\|\mathcal{W}\|_0}{\|\mathcal{W}\|} \geq s_{\mathcal{W}}} \mathcal{E}_{emp}(f(x, \mathbf{W})). \tag{1}$$

For simplicity, we presented Algorithm 1 that optimizes the sparsity of a given fully connected (FC) layer while maintaining the accuracy of the model. The weight matrix of an FC layer with n input nodes and m output nodes is stored as a matrix $\mathcal{W}$ of shape $n \times m$. The algorithm takes as input a pre-trained DNN model, the weight matrix of the FC layer to be compressed, the required sparsity constraint, and a small part of the training dataset of the original pre-trained DNN. Additionally, it has standard hyper-parameters such as the number of epochs, batch size, and learning rate. To reduce memory requirements, we can store the weight matrix as sparse by setting weights to zero conforming to the sparsity constraint. The algorithm follows the standard training procedure by selecting a batch of samples from the given data set and employing the forward pass (Lines 2-3). Then, it applies the projected gradient descent (lines 4-5). Firstly, the gradients concerning the weights of the FC layer are calculated, and the weight matrix is updated by moving in the direction of the negative gradient (line 4). Then, the weight matrix is projected onto the feasible set, i.e., the sparsity constraint is imposed on the FC layer via *hard thresholding* procedure. Note that, to minimize accuracy degradation, the layers following the FC layer are also re-trained jointly (not presented in Algorithm 1). For these layers, the compression is employed by considering their pre-recorded sparsity values, if any. Retraining the layers of pre-trained DNNs can often result in overfitting. To address this issue, SparseComp introduces a regularization term and freezes the layers preceding the currently compressed layer.

Hard thresholding procedure. To impose a sparsity constraint, two thresholds, an upper and lower limit, are calculated. The weights between these thresholds—ones with smaller absolute values—are set to zero. As shown in fig. 1, which summarizes the compression process for an FC layer, weaker weights are pruned to satisfy the sparsity constraint. However, the pruned weights may reappear in the next epoch of the re-training procedure. With iterative pruning and retraining, the weights become more stable over time. SparseComp employs an identical procedure to compress the filters and biases of convolutional layers. The main idea is imposing sparsity via hard thresholding, allowing SparseComp to apply a single compression method regardless of layer types in the model.

3.2 Global Early Exit for Pre-Trained Networks

gNet is a single network architecture that replaces multiple exit branches of the traditional early-exit architectures with a single global exit layer that supports anytime output from any layer of any pre-trained model. It introduces minimal memory overhead for early exit, which is crucial since memory is extremely scarce in batteryless embedded systems. This architecture not only significantly reduces the memory overhead of storing the parameters of the exit branches but also eliminates the need to store previous exit points output separately, which further reduces memory footprint.

3.2.1 Overview of gNet Instead of inserting one exit branch after each intermediate layer, gNet takes the output from all intermediate layers as input. Developing such a ubiquitous exit architecture comes with 2 unique challenges. (1) When exiting from an earlier exit point of the baseline DNN, the outputs of the following exit points are missing. Unlike traditional exit branches, where individual exit branches process the output from each exit point, gNet requires a single fixed dimension input to the global exit branch. Thus, there is a need to compensate for the missing information from the future exit points. (2) The dimension of the output from each layer of the DNN varies, making it challenging to concatenate them directly. Besides, simply flattening all layers and then concatenating them is insufficient as it may create a bias towards the earlier layers where the output dimensions are usually larger. Moreover, a simple concatenated vector will be extremely large to store and process and will require a large fully connected layer for classification, defeating the purpose of early exit.

Figure 2 shows the general overview of gNet, which addresses these challenges with three major components – (1) augmentation with zero-padding, (2) concatenation with pooling, and (3) classification with a linear layer. First, the augmentation with zero-padding handles the missing output from the layers yet to be inferred by padding them with zero to compensate for dynamic input length to the exit branch. Next, the concatenation with pooling resizes all intermediate output layers to a predefined size and concatenates them across the channel dimension. It also shrinks features across channel dimensions and extracts meaningful features when zero-padded features exist. Finally, during classification with a linear layer, the concatenated data passes through a fully connected layer and estimates which class the input belongs to. We describe the details of each of these components in the following sections.

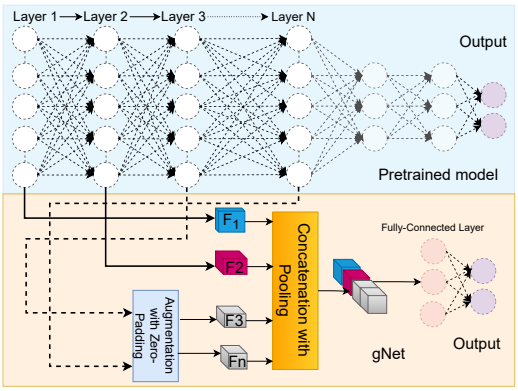

Figure 2: General overview of gNet. In this example, early exit occurs at layer 2; hence F_1 and F_2 are available, and the later features are zero-padded.

3.2.2 Augmentation with Zero-Padding The first challenge is that the output of later intermediate layers is not present while exiting from a previous layer. To compensate, we introduce the **Augmentation with Zero-Padding** layer, which performs zero-padding on the missing information from the future layers of the baseline DNN. Suppose the pre-trained network has n convolution layers and only exits from the convolution layer. Thus, there are n exits that are possible in maximum. When exiting from the i^{th} layer, where $i <= n$, then for 0 to i^{th} layer, we have valid intermediate outputs, and from $(i+1)^{th}$ to n^{th} layer, we add zeros of the required shape. The following equation represents the resultant feature vector:

$$FS = [\mathbf{F_1}, \mathbf{F_2}, ..., \mathbf{F_i}, \mathbf{0}, ..., \mathbf{0}] \quad (2)$$

Here FS is the feature set, $\mathbf{F_i}$ represents intermediate output from i^{th} layer. The shape of $\mathbf{F_i}$ is $C_i \times H_i \times W_i$ where C_i, H_i, and W_i stand for the number of channels, height, and width of the feature.

3.2.3 Concatenation with Pooling The second challenge is that the number of layers and the shape of the output tensor from each intermediate layer vary for different networks. Thus, developing a one-architecture-fit-all model is challenging. To address this, we accumulate the output of all intermediate layers to a fixed predefined shape using pooling (shown in Figure 2). This predefined shape is chosen based on the size of all intermediate outputs and the available memory of the microcontroller. We use 3D Maxpooling to convert each output vector to a predefined shape because 3D pooling preserves the channel information which is crucial for extracting important information. Simply using the same pooling parameters for all the layers is insufficient as the earlier layers' output tensor size has a larger height and width than the later layer counterparts and can create a bias towards earlier layers. We made all layers' output the same shape to reduce bias towards any specific layer on gNet by using larger pooling kernels on the earlier layers and smaller kernel sizes for later layers. However, we do not pool the channel dimension of all the layers to the same shape because the information content in the later layers is richer over multiple channels than the earlier ones. Therefore, we use pooling kernels of various sizes for each layer, where the kernel sizes are determined based on the memory of the target microcontroller. Once all the intermediate features are the same size, we concatenate them along the channel axis and feed this fused feature to the gNet.

3.2.4 Classification with a Fully-Connected Layer The classification layer consists of a FC layer that maps the flattened vector coming from the previous layer into the predefined classes. As the FC layer performs multiplication and accumulation (MAC) operations before going into the activation layer, the order of the flattened feature set does not matter here. Over the training phase, this layer learns to put more emphasis on the available *("real")* features and discard the zero-padded *("fake")* features. This emphasis on real features is attained by assigning larger weights to the real feature sections and smaller weights to zero-padded sections.

3.2.5 Agile Training of gNet gNet requires maintaining the output performance irrespective of the output availability from all the exit layers. To achieve this goal, we propose an agile training procedure. The gNet learns to adapt to different early exit scenarios through this training, which helps them generalize patterns, correlations, and characteristics in the different incoming data. We first develop a simulated training dataset reflecting the effect of an early exit. If a network has n layers to exit from, the probability of exiting from the i^{th} layer would be $\frac{1}{n}$. When exiting from the i^{th} layer, where $i <= n$, then pad zeros from $(i+1)^{th}$ to n^{th} layer. The training procedure of $gNet$ is shown in algorithm 2. Thus, the internal network parameters are iteratively updated depending on the discrepancies between predicted and actual outputs during training to maximize the network's performance for all existing conditions. Since all the exit scenarios are encountered in the training period, we reach a global model capable of handling different exit scenarios.

Algorithm 2: Training of gNet

Inputs: $\mathcal{M}$—pre-trained DNN model, $gNet$—generalized early exit model, $\mathcal{D}$— dataset of (X=value, Y=label) pairs.
Parameters: e—SGD epochs, B—batch size, η—learning rate, τ—validation threshold, $\mathcal{L}_{val-best}$–best validation loss.

repeat
  pick i^{th} exit layer randomly with probability of $\frac{1}{n}$ from n layers
  FS = `forward`$(\mathcal{M}, \mathcal{X})$ ▷ get intermediate results
  $\hat{Y}$ = `forward`$(gNet, FS)$ ▷ get predictions
  $gNet \leftarrow gNet - \eta \left(\sum_{i=0}^{B} \nabla_{gNet} \mathcal{L}_i(y_i, \hat{y}_i) \right)$
  $\mathcal{L}_{val} \leftarrow \mathcal{L}(gNet(FS_{val}), Y_{val})$
  if $\mathcal{L}_{val} < \mathcal{L}_{val-best}$ **then**
    save gNet
    $\mathcal{L}_{val-best} \leftarrow \mathcal{L}_{val}$ $count \leftarrow 0$
  else
    $count \leftarrow count + 1$
  if $count >= \tau$ **then**
    exit
  else
    continue
until *e epochs*

3.3 Energy-Aware Intermittent Execution

FreeML runtime executes ultra-tiny DNN models augmented with gNET layer-by-layer by incorporating a set of optimizations.

3.3.1 Buffer Reduction After executing a layer connected to an exit branch, FreeML performs an additional max-pooling before storing the output in a buffer. This step reduces the required buffer size by not saving the entire layer output and optimizes memory utilization while making the inputs of the early exit branch available at any

time. However, max-pooling introduces a tolerable processing overhead for these layers., which can be further reduced by in-position max-pooling while calculating the output layer.

3.3.2 Compressed Computation Next, when gNET is executed through an exit branch, the FreeML runtime uses a smart approach to save time and energy by avoiding computations for zero-padded inputs that belong to layers not yet executed. This method efficiently skips MAC operations for zero-padded inputs, leading to more efficient execution of early exit branches. Unlike traditional accelerators that only support structured compressed computing, microcontrollers allow unstructured compressed computation, presenting us with this unique opportunity.

3.3.3 Flexible Energy-Aware Execution Policies As a result, this strategy leads to different computational costs for each early exit in a DNN model. Specifically, the exit layers linked to later branches tend to have a higher computational overhead compared to the earlier exit branches. However, they offer enhanced accuracy in return. This characteristic provides a unique opportunity for employing different policies. For instance, the application can decide to continue DNN execution for optimal accuracy or stop by selecting an early exit branch that can output a result with reasonable accuracy and timing overhead. Pre-recorded time and energy overheads of each layer can determine which early exit layer to choose. This gives developers the flexibility to balance processing demands, energy efficiency, and accuracy based on their application's requirements.

4 Implementation

We implemented the proposed algorithms SparseComp and gNet in Python using the PyTorch framework.The adaptive and optimized model is automatically converted into C headers per layer including arrays holding the parameters of each layer. These platform-independent headers are combined with the ML libraries supported by existing intermittent computing runtimes, e.g., Alpaca [31] and ImmortalThreads [43], to be executed intermittently.

4.1 General Early Exit (gNet) Implementation

We create the model according to the description in section 3.2.1 and train it according to section 3.2.5. We use a stochastic gradient descent optimizer, Adam optimizer, with a default learning rate of 5×10^{-3}. While training, we monitored the validation loss (depending on the application and pre-trained network) to avoid overfitting. We saved the model if the validation loss decreased, and if the validation loss kept increasing for 10 consecutive epochs, we stopped the training. The best-saved model was used for the inference. To be consistent with prior works [34, 36], we add exit points after the convolutional layers only. However, our proposed gNet supports optimal exit points described in [23].

4.2 SparseComp Implementation

SparseComp is implemented as a plug-and-play tool that takes a Pytorch model and the different datasets needed for training (training and test sets). A validation set can optionally be used to evaluate the model during compression and identify the best one. In the case of overfitting, it is possible to specify a regularization term, which will be used in the backpropagation phases. SparseComp automatically excludes norm layers or biases from compression since they have a low number of parameters and can have huge impacts on accuracy. For gradient and backpropagation operations, SparseComp uses PyTorch's automatic differentiation engine Autograd, which allows compressing and retraining of any Pytorch model. In the case of models that require data preprocessing for the inferences or that use unconventional forward functions, it is also possible to inject both functions during the compression phase. The preprocessing function is applied on each input batch before the forwarding phase, while the forward function is applied on the input instead of the standard one. These features allow unconventional models, such as gNet, to be compressed without any changes.

4.3 Model Code Generation

After compressing and adding early exit layers, FreeML converts the model parameters from Python into C headers per layer. To store the weights of the model we used the CSR (Compressed Sparse Row) representation, which allows computationally efficient matrix operations. In CSR representation, a matrix is flattened and only the non-zero values are saved along with their column indices and extent of rows. It is also possible to choose different representations e.g., pair representation (index, value), CSC (Compressed Sparse Column), and COO (Coordinate Format) [16].

4.4 Intermittent Execution Runtime

There are recent works that provided basic ML library implementations that can be executed intermittently on MSP430FR [21] series MCUs. Therefore, we did not implement an ML library from scratch and relied on existing publicly available code targeting intermittent systems. For instance, Sonic [14] provided a task-based implementation of ML operations based on Alpaca [31]. Similarly, ImmortalThreads [43] has also a checkpoint-based ML library. We used and modified the ML library implementation of Sonic for ease of portability since the ImmortalThreads library had some platform-specific assembly code in its source. The SONIC ML library keeps the model parameters in non-volatile memory by using specific structures. We further process the C header outputs of SparseComp and make them compatible with the SONIC ML library. We implemented a small runtime on top of the Alpaca for Sonic to execute models layer-by-layer and perform runtime optimizations and decisions for gNet as mentioned in Section 3.3.

5 Evaluation

We consider neural network models of two sizes – (1) **tiny** models that are suitable for constrained embedded systems , and (2) **micro** models which can fit in extremely constrained systems. Table 2 shows the details of all networks. For training, we use batch normalization and RELU activation after each convolutional layer for all these models. To reduce overfitting, we use dropout between the fully connected layers. We train these networks on a GPU machine with two RTX 3090Ti. These models are considered the *pre-trained* models for our evaluation.

We aim to test the effectiveness of FreeML on datasets from different domains. Firstly, we conduct all our experiments on an image classification dataset (CIFAR-10 [25]), an acoustic dataset (Google Keyword Spotting (KWS)[37]), and a motion-based human activity recognition dataset (HAR [20]. We divide the training dataset into a train set and a test set. After shuffling all the training datasets, we

Table 2: Pre-trained DNNs considered in this section.

CIFAR-10[25]		**KWS**[37]		**HAR**[20]	
Micro	**Tiny**	**Micro**	**Tiny**	**Micro**	**Tiny**
C:64 × 3 × 3 × 3	C:32 × 3 × 3 × 3	C:16 × 1 × 3 × 3	C:16 × 1 × 3 × 3	C:32 × 3 × 1 × 12	C:32 × 3 × 1 × 12
C:128 × 64 × 3 × 3	C:32 × 32 × 3 × 3	C:32 × 16 × 3 × 3	C:32 × 16 × 3 × 3	C:32 × 32 × 1 × 12	C:32 × 32 × 1 × 12
C:64 × 128 × 3 × 3	C:64 × 32 × 3 × 3	C:64 × 32 × 3 × 3	C:64 × 32 × 3 × 3	C:64 × 32 × 1 × 12	C:64 × 32 × 1 × 12
F:256 × 256	C:64 × 64 × 3 × 3	F:2304 × 64	C:64 × 64 × 3 × 3	F:1600 × 128	C: 64 × 64 × 1 × 3
F:256 × 64	C:128 × 64 × 3 × 3	F:64 × 10	F:256 × 128	F:128 × 128	C:128 × 64 × 1 × 12
F:64 × 10	C:128 × 128 × 3 × 3		F:128 × 64	F:128 × 6	C:128 × 128 × 1 × 12
	F:2048 × 128		F:64 × 10		4× F:128 × 128
	3× F:128 × 128				F:128 × 6
	F:128 × 10	C: Convolution Layer	F: Fully Connected Layer		

put 90% data into the train set and the remaining 10% in the validation set. The test dataset remains unseen throughout the training sessions and is only used for inference.

5.1 Evaluation of Compression (SparseComp)

5.1.1 Performance Metrics. To evaluate the performance of SparseComp we report two metrics: accuracy and compression rate. Accuracy is used to compare the performance of a model before and after compression. The compression rate indicates how significant the pruning is, and it is calculated by dividing the original number of parameters by the number of pruned parameters.

Table 3: Comparison between Genesis and SparseComp

Network	**Original Model**		**Genesis**		**SparseComp**	
	Acc.	Size.	Acc.	Size.	Acc.	Size
Image Classification (MNIST)	99.06%	1905.3 kb	99%	34.6 kb (55×)	98.6%	19.8 kb (95.8×)
Human Activity Recognition (HAR)	91.93%	2100, 6 kb (8.2×)	88,0 %	256.8 kb (8.2×)	88,55 %	29.8 kb (70.3×)
Google Keyword Spotting (KWS)	79.97%	1316,5 kb	84,0%	215.5 kb (6×)	75.98%	59.2 kb (22.2×)

5.1.2 Comparison against GENESIS We used Genesis [14], the de facto model optimization tool in intermittent computing, as a baseline. SparseComp uses a simple approach for each model and each layer compared to Genesis, which employs a NAS-oriented approach with separation and post-facto pruning. We considered the same datasets and model structures reported by authors of Genesis [14, Table 2]. Table 3 shows the size and accuracy of the uncompressed original models as well as the overall accuracy and size together with the compression rate of the compressed versions. On the MNSIT and HAR datasets, SparseComp achieves significantly better compression while maintaining similar precision. For KWS, even our uncompressed original model could not reach 84% accuracy of the compressed model reported by the authors of Genesis, prevented us from making a sound comparison against the Genesis KWS model. When we fixed the size of our compressed KWS model to the same size as the compressed KWS model in Genesis, we achieved an accuracy of 79,97, which is 4% less than the accuracy reported by Genesis. In this case, the accuracy drop from the uncompressed model was only 1 %. We further compressed the model and observed that SparseComp can compress the original model almost 22 times smaller with under 4% accuracy drop. Our evaluation showed that SparseComp is a significantly better and simpler approach compared to existing solutions.

5.1.3 General Evaluation We evaluated SparseComp using the models shown in Table 2. We would like to highlight that SparseComp can be applied to compress different models and DNNs without any modification. Figure 3 represents the accuracy of the models over several compression iterations. At each step, we define a memory size constraint, and SparseComp compresses the models to meet that requirement. We observed that for most of the models, SparseComp obtains promising results with a high compression rate. The accuracy drop is almost linear even reaching kB-sized models. Since SparseComp performs re-training, at some iterations, we even observed better accuracy with a smaller model. It is worth mentioning that uncompressed models with higher accuracy lead to higher and more stable accuracy during the compression.

5.1.4 Effect of Available Training Samples on SparseComp. In this section, we evaluate the impact of using only a subset of the dataset on the compression performance. Based on Figure 4, SparseComp can compress a pre-trained model up to 64 kb without significantly affecting accuracy, even when using only a fraction of the training data. However, when compressing to 32 kb, the performance loss is more noticeable, and a larger pool of training examples is more effective. It's worth noting that the dataset we considered initially had a small number of examples. With larger datasets, the percentage of usage can be even lower.

5.2 Global Early-Exit (gNet) Evaluation

We compare gNet with a variant of state-of-the-art e-Perceptive [34], where we do not retrain the baseline model (compressed with SparseComp) for a fair comparison and call this Not Re-Trained e-Perceptive (NRT-eP). Finally, we provide an ablation with a re-trained baseline model (BancyNet [36]).

5.2.1 Performance Metric. We used memory, throughput, and performance as metrics to evaluate the performance of gNet from two aspects. First, we compare the **number of required parameters** between the baseline algorithm and gNet, which translates to the model's memory requirement. Next, we measure the time required to infer the model to quantify throughput. Instead of reporting the absolute time, we report the **normalized inference time** for a more fair and device-agnostic comparison. To calculate the normalized inference time, we normalize the inference times against the inference time of the pre-trained models, which we consider as 1. Smaller normalized inference time indicated high throughput and less runtime overhead. Finally, to assess the performance, we report the model's **accuracy**.

5.2.2 Memory and Accuracy Trade-Off. Baseline individual early-exit models (NRT-ep) store n number of different exit models for a n layered DNN, which makes it inefficient in terms of memory overhead. On the other hand, gNet is a single standalone model that works regardless of the number of layers in the baseline model. This

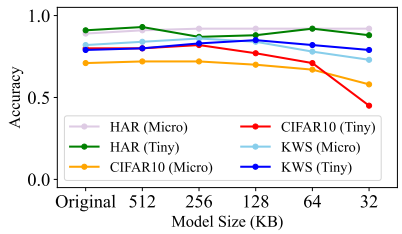

Figure 3: Accuracy of compression.

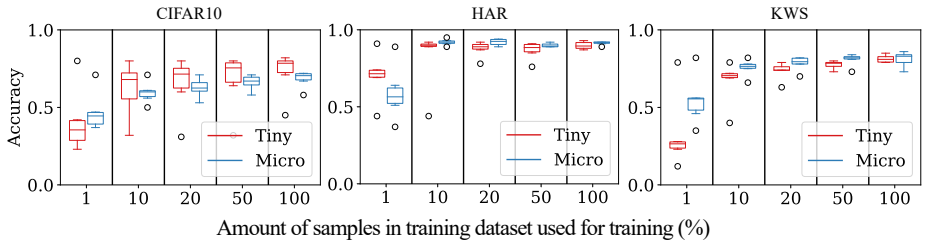

Figure 4: Effect of training dataset percentage on SparseComp compression.

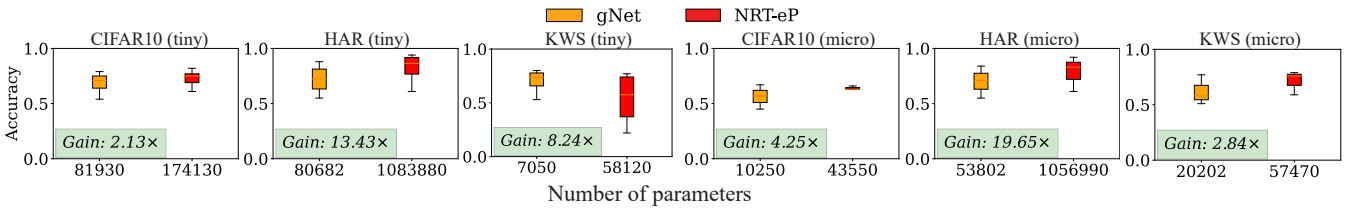

Figure 5: No. of parameters for gNet and NRT-eP. gNet requires fewer parameters than NRT-eP, reducing memory overhead.

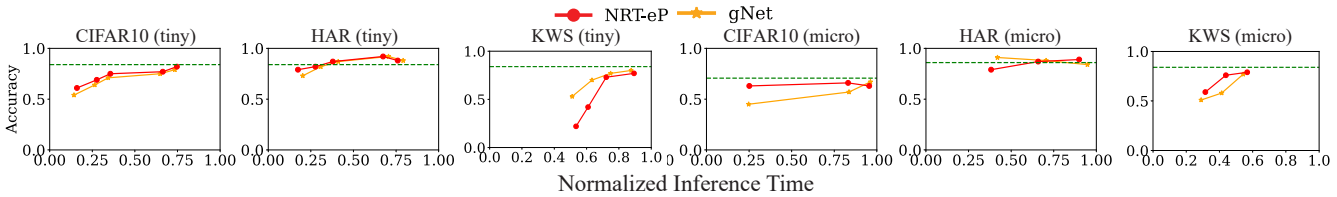

Figure 6: Accuracy and normalized inference time comparison of gNet and NRT-eP with 6 pre-trained models and 3 datasets. Here, the dashed green line shows the baseline models' accuracy.

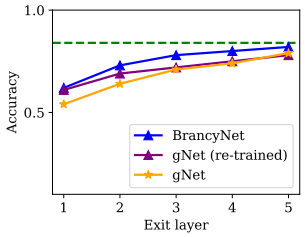

Figure 7: Effect of retraining the baseline on gNet.

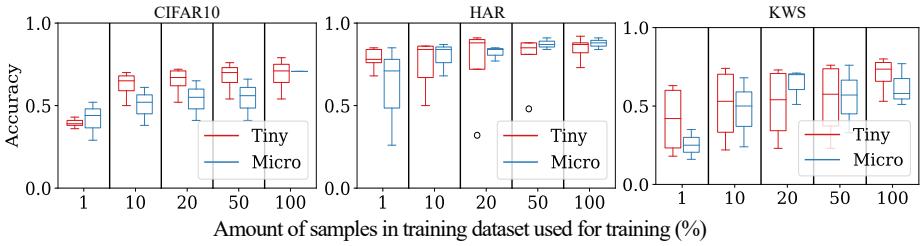

Figure 8: Effect of training dataset percentage on gNet with CIFAR-10, HAR, and KWS dataset.

makes the gNet memory efficient and more suitable for small and edge device-level deployment. Figure 5 shows that gNet requires 2.13−19.65 times fewer parameters than NRT-ePs while maintaining the accuracy distributions at different existing layers. The models with the HAR dataset have the highest memory overhead reduction compared to the other datasets due to the smaller input size of baseline HAR models (tiny, micro). As the input shape is smaller, the intermediate feature sizes are smaller, too. Additionally, after passing the augmentation with a zero-padding layer, the feature shapes get a further reduction. This enables us to design a small gNet model, which results in a very high gain in terms of parameters. This satisfies our claim of one-model-fit for all.

5.2.3 Inference Time and Accuracy Trade-Off. Figure 6 shows the accuracy vs the normalized inference time required for all dataset and pre-trained model configurations described in Table 2. For all three tiny models, gNet achieves 90^{th} percentile accuracy gain of $0.12 - -0.31\%$ while reducing the inference time by $2.41\% - 3.17\%$. However, for micro models, this accuracy gain reduces to $0.12 - -0.31\%$ with a $2.52\% - 4.85\%$ reduction in inference time. Due to the larger depth of the tiny baseline models, the global exit branch of tiny models has a larger input size, and thus, the reduction of inference time is lower than in micro models. However, these more comprehensive intermediate features result in higher accuracy gain. On the other hand, micro baseline models are smaller, generating smaller intermediate features as input to the exit branch, and thus, we experience a higher reduction of inference time. In summary, along with reducing the memory overhead by up to 19.65%, we reduce inference time with negligible to no accuracy loss.

5.2.4 Comparison with Jointly-Trained BranchyNet. Figure 7 compares gNet with the popular jointly trained early exit method, BranchyNet [36] on the tiny model with CIFAR-10 dataset. BrancyNet jointly trains the exit branches are trained alongside the

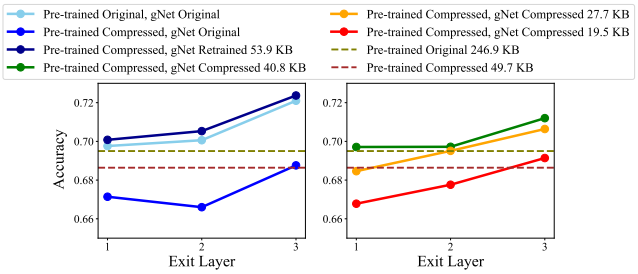

Figure 9: Compression of Early Exit. gNet benefits and performance are maintained during compression.

baseline model. Though BranchyNet gains accuracy by at least 3.65% compared to gNet, it required 4.42 times fewer parameters than BranchyNet. We have also jointly retrained gNet, and the accuracy increases by up to 6.5% than gNet.

5.2.5 Effect of Available Training Samples. This section evaluates the effect of available training data samples required to train gNet. This analysis is crucial as, in real-world scenarios, the full training dataset may not be available to train gNet. Figure 8 shows that the accuracy of all models increases with the availability of more samples in the training set. However, the gain from 10% available samples and 100% samples is not that significant. The median accuracy gain from 10% sample to 100% sample is only 4.0% – −20.49% for tiny models and 4.00% – −19.99% for micro models over all datasets. However, for micro models, this average gain is much higher as intermediate features of micro are less comprehensive than tiny models, gNet can achieve higher accuracy gain. More complex datasets, e.g., KWS, require more training data to reach higher accuracy, and thus the accuracy gain between 10% to 100% training samples is up to 20.49%.

5.2.6 Evaluation of Early-Exit Compression We used the CIFAR10 dataset with the aim of compressing models to meet the memory requirements for the MSP430FR device. The compression targets both the baseline model and the gNet with, starting sizes of 246.9 KB and 53.9 KB, respectively. The first step is to compress the baseline model, SparseComp is able to shrink the sizes to 49.7 KB with 1% accuracy loss. Figure 9 on the left shows the performance of gNet when relying on the original or the compressed baseline model for the intermediate results. Without a retraining phase, the gNet achieves accuracy comparable to the base model. When it is retrained, however, it even surpasses the performance of the original uncompressed models. On the right, instead, are shown different compressions applied on gNet. The intermediate results are provided by the compressed model and it can be seen that the key features of gNet are maintained during the compression phase by SparseComp. The two compressed models outperform the base model and reach the accuracy of the uncompressed versions, with a total size reduced from 300.8 to 77.4 KB allowing their use on the memory-constrained MSP device.

5.3 Model Execution on Real Hardware

We evaluated tiny versions of the gNet models for Cifar-10, HAR, and KWS using MSP430FR5994 [21], the defacto MCU in batteryless systems, configured to operate at 1 MHz. We report the memory overhead of these models and their total energy consumption and

Table 4: Time and energy consumption during the continuous and intermittent execution on MSP430FR5994.

Network	Exec. Time (sec)			Energy Cons. (mJ)		
	Cont.	Int.	E.E.	Cont.	Int.	E.E.
Cifar-10	139.5	166.2	156.1 (E3)	264.9	320.1	297.3 (E3)
HAR	74.6	83.4	75.8 (E5)	135.8	152.3	137.2 (E5)
KWS	184.6	218.7	185.8 (E2)	355.2	419.9	356.5 (E2)

Table 5: Total memory overhead of FreeML and tiny models

	FreeML	Models		
		Cifar-10	HAR	KWS
.text	12.5 kb	376 Byte	362 Byte	334 Byte
RAM	4 Byte	0	0	0
FRAM	12.8 kb	200.9 kb	30.4 kb	219.4 kb

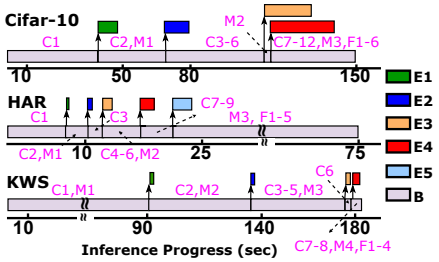

Figure 10: Time overheads of the inference layers on continuous power (C:Convolution, F:Fully Connected, M:Maxpool).

execution time under continuous and intermittent energy supply. For energy harvesting, we used Powercast TX91501-3W-ID power transmitter and P2110-EVB power harvester that includes a 1mF onboard energy storage supercapacitor.

5.3.1 Time and Energy Performance. Table 4 shows the execution time and energy consumption of these models on continuous power ("Cont." column) as well as under intermittent power *without* exit branches ("Int" column), and *with* early exit branches ("E.E." column). Our results indicate that early exit layers considerably improve the time and energy overheads of intermittent execution (as evident from the "Int" and "E.E." columns of Table 4). For intermittent execution with exit branches, we used the time it takes to run a model continuously as a time constraint to trigger anytime outputs to keep things simple. The idea was to make our models output predictions as fast as their execution on continuous power. When executing HAR and KWS, FreeML runtime used exit branches 5 (E5) and 2 (E2) in these models, respectively, significantly improving their intermittent execution times. For Cifar-10, FreeML used a relatively earlier exit branch 3 (E3), which led to intermittent execution of the early exit branch taking even less time than intermittent execution without using early exits. Figure 10 provides a detailed view of the time overheads of each individual layer on continuous power. It shows that the execution times of early exit branches are relatively smaller than those of later exit branches, which is consistent with our arguments in Section 3.3.

5.3.2 Harvested Power and Early Exits. To observe how ambient power affects the exit branch taken, we emulated the energy harvesting process of the Powercast receiver to charge a 1mF onboard supercapacitor with different input power levels. We considered the intermittent execution of our models with early exit branches and imposed the same time constraint as in Section 5.3.1. Figure 11

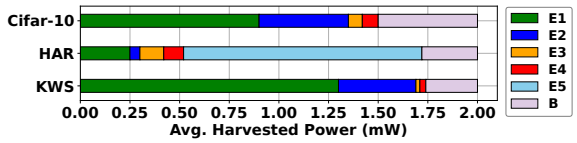

Figure 11: Selected exits based on the average harvested power for the 1mF capacitor of the P2110-EVB RF harvester.

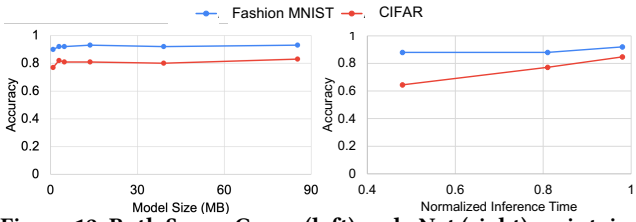

Figure 12: Both SparseComp (left) and gNet (right) maintains the performance for deeper ResNet34 network.

presents the exit branches taken with average input power values ranging from 0 to 2 mW. With Cifar-10, FreeML runtime triggers the first exit when the average input power is under 0.85mW to produce results within the time constraint. As the input power increases, model layers are executed faster due to faster charging time, and FreeML runtime takes later exit branches for better accuracy. When the average output power is higher than 1.5mW, the latest exit branch is no longer beneficial since the rest of the model can be executed faster, and the best accuracy can be obtained. Similar behaviors can also be observed for KWS and HAR models.

5.3.3 Memory Requirements Table 5 shows the memory requirements of our deployed models on MSP430FR5994 equipped with a 256KB FRAM and 8KB SRAM. As seen from this table, SparseComp has effectively reduced the memory footprint of machine learning models, enabling them to fit in memory-constrained devices. FRAM allocation was always within the limit available to the device, which depends on the number of parameters, the layer's input/output sizes, and the working buffers required to execute models intermittently.

5.4 Beyond Intermittent Computing

Though we focus on batteryless devices in this work, we envision that the benefit of our proposed SparseComp and gNet can go beyond intermittent computing and batteryless devices without any modification. We evaluate these algorithms on deeper ResNet34 [18] pre-trained models with two datasets (CIFAR-10 and FashionMNIST [38]) suitable for mobile edge devices to validate our vision. We assess the model's effectiveness using the normalized inference time, the number of parameters (described in Section 5.2.1), and accuracy. SparseComp applies pruning to the Residual Block in the same way it does on conventional networks, making it a one-for-all pruning method. Figure 12(left) shows that SparseComp compresses the models up to 94 times without a significant drop in accuracy. Figure 12(right) demonstrates that gNet reduces the inference time by 2%-52% and memory-overhead by 7.86 – 3.93 times without significantly sacrificing the accuracy.

6 Discussion and Future Work

Other Platforms. We tested deploying compressed models to Apollo 4 Blue Plus which has an ARM-Cortex M4 with a floating point unit and comes with an internal 2 MB MRAM (Magnetic Random-Access Memory) thus providing an opportunity to speed up complex computations. We were able to successfully deploy our model on Apollo 4 using Ambiq's NeuralSPOT [5] AI enablement library. ML model deployment using a software development kit prioritizes ease of use over runtime cost and memory footprint optimization, resulting in computational overhead. Even with the bare metal version, software support is required to save and restore volatile data onto MRAM at runtime to ensure correct resumption after power failure as, unlike MSP430, read and write variables are not memory mapped and require explicit APIs for read/write access; thus adding significant overhead to the execution time. In the future, we plan to explore this aspect more to show an end-to-end evaluation of our technique on the Apollo4 platform.

Support for More Complex Models. Comparatively hard datasets, e.g., CIFAR-100 require deeper and more complex networks (including residual edge or recurrent layers) making them less suitable for intermittent devices. SparseComp can compress these networks but when they reach the desired memory footprint, the accuracy drop is significant due to the high number of pruned parameters. We only used unstructured pruning to avoid changing the model architectures, even though structured pruning is required to reduce their runtime buffer requirements. In future work, we plan to explore the performance of SparseComp in structured pruning as well.

Retraining the Baseline Model during Compression. While SparseComp works on pre-trained models, it is also possible to retrain all model layers at each compression iteration. We can also train a model from scratch during the compression process. This can be very effective for simple datasets, but for more complex datasets, e.g., KWS, we found out that it is better to use pre-trained models as it training converges with higher compression rates.

Reducing Input Size to gNet. One of the shortcomings of gNet is the comparatively large input to the global exit branch. Though we reduce the MAC operation and even memory overhead during our MCU implementation by ignoring multiplication with zeros, the input size to gNet is larger for the deeper layers. In the future, we plan to implement attention-based max-pooling which can successfully pool only the informative components from the output of each layer making the input to the exit-branch even smaller. Moreover, we will be able to better differentiate between the 'real' and 'zero-padded' components in the earlier layers, resulting in a higher accuracy for the earlier layers. This paper focuses on fine-grained early-exit with exit points after each convolution layer, but our proposed gNet can be extended to coarse-grain early-exit with exit points after selected layers [23]. Besides, we can further add exit points after fully connected layers if the execution time of inferring the fully connected layers is higher than the exit layer.

7 Conclusion

We proposed FreeML, a novel framework that complements a pre-trained ML model with early exits to react to the power availability compresses the model to fit into a modest memory footprint, and generates C code for inference to run on a resource-constrained embedded sensing devices sensor node. Our evaluation shows that FreeML can achieve a 95× compression rate on pre-trained DNN models reducing their size from MBs to KBs so they can easily be deployed on embedded sensing platforms, as demonstrated by

our deployment on MSP430FR5994. Furthermore, FreeML achieves memory overhead 59× while requiring 65% lesser time for inference while bearing a minimal drop in accuracy.

Acknowledgments

This paper is 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. This paper was also supported, in part, by NSF grants CNS-2347692.

References

[1] 2024. FreeML Github Repository. https://github.com/tinysystems/FreeML.
[2] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 70–81.
[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 Kasim Sinan Yildirim. 2022. AdaMICA: Adaptive Multicore Intermittent Computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 3 (2022), 1–30.
[5] Ambiq. 2022. Ambiq Accelerates the Development of Optimized AI Features with neuralSPOT. https://ambiq.com/news/ambiq-accelerates-the-development-of-optimized-ai-features-with-neuralspot/.
[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*. 207–221.
[7] Abu Bakar, Tousif Rahman, Rishad Shafik, Fahim Kawsar, and Alessandro Montanari. 2022. Adaptive Intelligence for Batteryless Sensors Using Software-Accelerated Tsetlin Machines. In *Proceedings of SenSys*.
[8] Abu Bakar, Alexander G Ross, Kasim Sinan Yildirim, 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.
[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 Network Sensor Systems*.
[10] 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 (TODAES)* 27, 3 (2022), 1–50.
[11] 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*.
[12] 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 (TECS)* (2022).
[13] Analog Devices. 2020. Artificial Intelligence Microcontroller with Ultra-Low-Power Convolutional Neural Network Accelerator. Retrieved June 15, 2022 from https://datasheets.maximintegrated.com/en/ds/MAX78000.pdf
[14] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–213.
[15] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. Protonn: Compressed and accurate knn for resource-scarce devices. In *International conference on machine learning*. PMLR.
[16] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[17] 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).
[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. arXiv 2015. *arXiv preprint arXiv:1512.03385* 14 (2015).
[19] Shawn Hymel, Colby Banbury, Daniel Situnayake, Alex Elium, Carl Ward, Mat Kelcey, Mathijs Baaijens, Mateusz Majchrzycki, Jenny Plunkett, and other. 2022. Edge Impulse: An MLOps Platform for Tiny Machine Learning. *arXiv* (2022).
[20] Andrey Ignatov. [n. d.]. HAR. https://github.com/aiff22/HAR
[21] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf
[22] Bashima Islam and Shahriar Nirjon. 2020. Zygarde: Time-Sensitive On-Device Deep Inference and Adaptation on Intermittently-Powered Systems. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3.
[23] 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 Annual International Conference on Mobile Systems, Applications and Services*. 42–55.
[24] 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* (2015).
[25] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).
[26] 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 Conference on Embedded Networked Sensor Systems*. 138–152.
[27] 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*. 175–190.
[28] Seulki Lee and Shahriar Nirjon. 2022. Weight Separation for Memory-Efficient and Accurate Deep Multitask Learning. In *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 13–22.
[29] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.
[30] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270* (2018).
[31] 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.
[32] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)* 16, 1 (2020), 1–24.
[33] Hashan Roshantha Mendis, Chih-Kai Kang, and Pi-cheng Hsiu. 2021. Intermittent-aware neural architecture search. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–27.
[34] 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 Conference on Embedded Networked Sensor Systems*. 382–394.
[35] Shinichi Nakajima, Masashi Sugiyama, and S Babacan. 2011. Global solution of fully-observed variational Bayesian matrix factorization is column-wise independent. *Advances in Neural Information Processing Systems* 24 (2011).
[36] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, 2464–2469.
[37] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).
[38] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747
[39] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. 2018. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. 278–291.
[40] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. 2017. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–14.
[41] 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*. 41–53.
[42] Eren Yildiz, Saad Ahmed, Bashima Islam, Josiah Hester, and Kasim Sinan Yildirim. 2023. Efficient and Safe I/O Operations for Intermittent Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 63–78.
[43] Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. 2022. Immortal Threads: Multithreaded Event-driven Intermittent Computing on {Ultra-Low-Power} Microcontrollers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 339–355.