UnIT Pruner: Unstructured Inference-Time Pruning for Battery-Free Systems

Abstract

With deep neural networks (DNNs) being increasingly deployed on low-resource devices, achieving efficiency without sacrificing accuracy has become a critical challenge. Model pruning, commonly performed during training, is a common approach to reducing model size and computational cost, even in environments where computational resources are not limited. However, traditional pruning methods are inherently static, often leading to accuracy losses that hinder their applicability in dynamic, real-world settings. Although some inference-time pruning techniques exist [23, 26], these methods are generally structured, introducing similar accuracy compromises as conventional pruning methods. In this paper, we propose a novel, unstructured pruning algorithm that performs adaptive, input-specific pruning during inference: Unstructured Inference-Time Pruner (*UnIT Pruner*). Unlike traditional approaches, *UnIT Pruner* dynamically skips redundant operations in real time based on the unique properties of each input, allowing for efficient computation without significant accuracy trade-offs. Our algorithm can complement existing methods, enhancing the balance between energy efficiency and accuracy and, in certain systems, even reducing latency. Experimental results show that our method reduces MAC operations by 70.38–87.39% with as low as 0.43% accuracy drop. Additionally, we achieve 74.4–94.7% less inference time and consume 74.2–96.5% less energy on microcontrollers when compared to other pruning methods. Our proposed algorithm achieves state-of-the-art inference-time pruning performance, demonstrating its effectiveness in deploying DNNs in resource-constrained environments.

Keywords

Battery-free systems, Unstructured pruning, Inference-time pruning, Fast approximate division, MAC operation

1 Introduction

As artificial intelligence advances, deploying deep neural networks (DNNs) on edge devices is transforming how data is processed and interpreted directly at its source. Resource-constrained microcontroller units (MCUs) are critical to this shift, enabling real-time applications such as healthcare monitoring [8], environmental sensing [12, 20], and industrial automation [40] in settings where constant connectivity and power are not guaranteed. However, the demand for energy-efficient computation becomes even more pronounced with the emergence of battery-free, energy-harvesting devices. These ultra-low-power MCUs, which harvest energy from sources like ambient light, vibrations, or radio frequencies, offer the possibility of long-term, maintenance-free operation but present strict limitations on power availability and computational resources.

Inference in Deep Neural Networks (DNNs) on battery-free devices presents unique challenges due to the combination of limited memory, low computational power, and the need to operate with minimal energy harvested from ambient sources [11, 19]. These devices must manage both resource constraints and the intermittent nature of power availability, which often leads to frequent power loss. As a result, data must be saved in persistent memory regularly to avoid loss, which adds significant overhead to the system [16, 32]. This intermittent power loss, coupled with the limitations of available memory and computational resources, creates significant hurdles in maintaining efficient and accurate performance in real-time applications on battery-free devices.

Various studies have proposed DNN deployment techniques to reduce memory footprint and computational overhead, allowing these models to run efficiently on edge devices with limited memory and power. Techniques such as train-time pruning [29], quantization [43], specialized pruning [14], tensor decomposition, and early-exit strategies [9, 19, 34, 39]. Other research focuses on compressing DNNs to minimize execution time and energy consumption. Several intermittent systems have adopted similar strategies on shallower, smaller DNNs to maintain accuracy, including neural architecture search (NAS)-based approaches that generate various compressed configurations of a DNN [11].

A major bottleneck in DNN inference on these devices is the high cost of multiply-accumulate (MAC) operations. To illustrate, in a MSP430 [3], a multiplication operation requires 77 cycles [1] compared to 6 cycles [3] required for addition. Thus, reducing the number of MAC operations is crucial to making DNNs feasible on batteryless systems, especially for low latency and high energy efficiency applications.

Thus, among these techniques, pruning is valuable, not only for reducing the model size but also for eliminating redundant MAC operations, thereby enhancing the power efficiency of the system without sacrificing accuracy. By strategically pruning the network, unnecessary computations are minimized, making the model more lightweight and energy-efficient while still maintaining high performance

under fluctuating power conditions. All pruning on battery-free devices is typically performed at training time, as most methods focus on optimizing the model before deployment [4, 24, 38]. However, training-time pruning methods often lack the flexibility to adapt to real-time conditions, such as changes in available energy or computational load, which limits their effectiveness in the dynamic environments of battery-free systems.

Only a few works have explored inference-time pruning [6, 13, 35], but they are not targeted at battery-free devices or other resource-constrained microcontrollers. These approaches often employ structured pruning, which does not take into account the unique characteristics of microcontroller units (MCUs), such as the lack of floating-point operations and single-threaded processing. Furthermore, structured pruning introduces high computation and memory overhead, making these methods unsuitable for battery-free systems. Additionally, current inference-time pruning methods are often limited by fixed pruning schedules or static criteria that fail to account for variable computational demands and power fluctuations in real-world environments. These methods lack the flexibility to adjust pruned elements adaptively, which can lead to unnecessary accuracy loss or underutilization of available resources. A more adaptable technique is needed to deploy DNNs on batteryless systems effectively—one that can bridge the accuracy gap while satisfying extreme memory and power constraints and that integrates adaptive pruning mechanisms responsive to real-time conditions, balancing efficiency and accuracy dynamically.

This paper proposes *UnIT Pruner*, the first algorithm of its kind, introducing novel unstructured inference-time pruning, specifically tailored to the unique device-specific constraints to enhance computational efficiency and scalability across a wide range of *resource-constrained* devices without compromising accuracy. It dynamically skips redundant MAC operations based on the specific characteristics of each input, providing a flexible and efficient pruning solution without predefined patterns.

UnIT Pruner first introduces a unique fine-grain pruning which neither prunes activation values nor weights but rather individual *connections*, enabling selective removal of redundant MAC operations throughout the network. It hinges on the relatively low cost of branching compared to MAC operations on devices without multi-stage instruction pipelining and/or dedicated multiplication hardware, allowing us to apply a dynamically generated threshold during runtime. *UnIT Pruner* uses a dynamic connection skipping mechanism, which selectively activates or deactivates connections based on each input's contribution to the final output. This approach reduces the number of MAC operations required to compute each forward pass, dynamically adjusting the model's computational load in real time.

Besides, *UnIT Pruner* preserves activations and weights that may be useful later available to the network, which is impossible for train-time pruning or structured inference time pruning as both are unable to dynamically skip individual connections. This unstructured approach increases model adaptability, allowing it to retain more nuanced information and perform efficiently across diverse datasets and tasks.

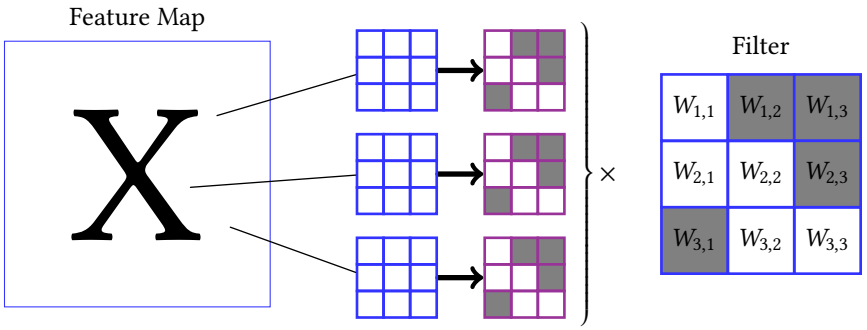
To reduce the computational overhead of pruning, *UnIT Pruner* offers three division estimation algorithms specifically designed to optimize the algorithm's overhead for varied devices and quantization methodologies. Each algorithm is customized to evaluate pruning impacts in real-time, balancing computational load with performance goals. The algorithms can be adapted to device, model, and task-specific constraints and particulars, enabling faster runtime and lowering the power and memory demands, making our approach suitable for resource-limited devices. While MCUs have this strict hardware limitation, high-performance GPU and CPU can overcome this limitation and show negligible performance improvement.

By introducing this first-of-its-kind unstructured pruning algorithm, we set a new standard for flexibility and efficiency in model compression. Additionally, our algorithm is compatible with existing pruning methods, further enhancing the trade-offs between energy efficiency, accuracy, and latency.

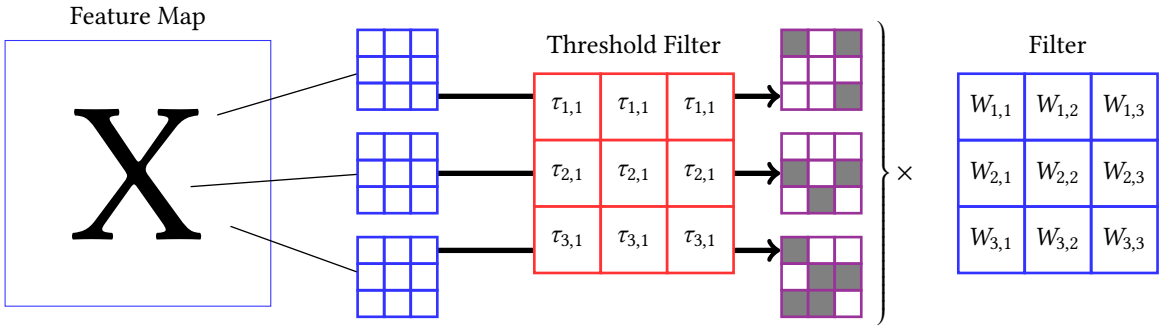
Our key contributions are as follows:

- We introduce *UnIT Pruner*: a novel unstructured inference-time pruning algorithm with *extremely low memory overhead* that prunes individual connections instead of activations or weights, enabling selective MAC operation removal across the network. Connections are activated or deactivated based on input relevance, dynamically adjusting computational load in real-time and leveraging Location-Specific Thresholding suited for devices without advanced pipelining or multiplication hardware.
- Our method retains activations and weights for potential future use, increasing adaptability and performance across varied tasks and datasets, amplifying energy savings, and maintaining accuracy in real-world, low-power applications.
- We present three tailored division estimation algorithms to minimize pruning overhead, adapting them to device, model, and task constraints. These algorithms enable faster runtime, lower power consumption, and reduced memory use, making *UnIT Pruner* ideal for constrained devices.

This work fills a critical gap by enabling adaptive, efficient DNN inference on batteryless MCUs, offering a robust and scalable solution for the next generation of intelligent, low-maintenance edge devices.



(a) Standard unstructured train-time pruning for convolutional layers removes parts of filters.



(b) *UnIT Pruner* does unstructured pruning in a different way for convolutional layers, pruning *individual* connections as opposed to entire filter components, similar to contemporary unstructured pruning methods for linear layers.

Figure 1: The Xs are feature maps or channels from a previous layer's output. With existing methods, the samples have some values ignored due to the way that unstructured train-time pruning works for convolutional layers. With *UnIT Pruner*, samples are taken from this feature map and passed through a *threshold filter* before being multiplied with the filter. This form of pruning is *edge* pruning.

We perform a rigorous evaluation using 4 datasets on an MSP430FR5994 [3]. Our results demonstrate that our method consumes 74.2% to 96.5% less energy compared to other pruning methods. Additionally, it achieves a 74.4% to 94.7% reduction in inference time. *UnIT Pruner* reduces MAC operations by 70.38% to 87.39%, with an accuracy drop as low as 0.43% when compared to the unpruned models.

2 Unstructured Inference-Time Pruning

We propose *UnIT Pruner*, an unstructured inference-time pruning algorithm that prunes individual connections between two nodes of a neural network with minimal overhead. *UnIT Pruner* consists of three key elements—(1) Dynamic Edge Pruning, (2) Location-Specific Thresholding, and (3) Fast Division Approximation.

2.1 Dynamic Edge Pruning

Traditional unstructured pruning is performed during training, and individual multiplications are skipped by setting values in the weight tensor to 0. Multiplications are skipped when evaluating dot products for those layers. Given an input matrix $\mathbf{X}^{m\times n}$ from the $(k-1)^{th}$ layer and a weight matrix $\mathbf{W}^{n\times p}$ from the k^{th} layer, the output $\mathbf{Y}^{m\times p}$ can be represented as

$$\mathbf{Y}_{i,j} = \sum_{l=1}^{n} \mathbf{X}_{i,l}\mathbf{W}_{l,j} \tag{1}$$

Where i indicates the row of $\mathbf{Y}$ and $\mathbf{X}$, j indicates the column of $\mathbf{Y}$ and $\mathbf{W}$, and $l \in [1, n]$. Summing the products of the corresponding elements in row i of $\mathbf{X}$ column j of $\mathbf{W}$.

In weight pruning, if $\mathbf{W}_{l,j}$ is an extremely low value, the multiplied output $\mathbf{Y}$ becomes insignificant, allowing skipping of the multiplication. On the other hand, activation pruning evaluates the magnitude of $\mathbf{Y}$ and removes further connections depending on this $\mathbf{Y}$.

However, with aggressive train-time pruning, multiplications that should *not* be skipped get skipped and result in increased error. Train-time weight pruning removes small values of $\mathbf{W}_{l,j}$ (10^{-3}), which can still be significant with

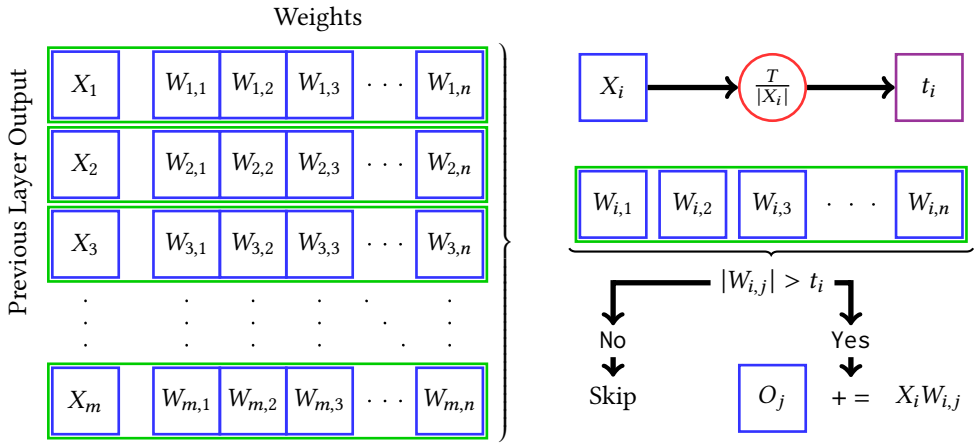


Figure 2: X is the vector of outputs from the previous layer of size m. W is the weight tensor that connects the previous layer to the next layer with n neurons. The next layer has its pre-activation values put into O which is initialized to all 0s. For each output X_i from the previous layer, a threshold $\frac{T}{|X_i|}$ is generated. Then, for each weight associated with X_i, it is skipped if its magnitude is below the threshold.

higher order values of $\mathbf{X}_{i,l}$ (10^4). Although train-time activation pruning aims to address this, it fails when the training and inference-time data distribution changes because the pruned connections are *static* due to being decided *during training*. To address this, *UnIT Pruner* introduces **D**ynamic **E**dge **P**runing (*DEP*), which preserves weight values that train-time pruning methods do not and dynamically skips multiplications based on the values of $\mathbf{X}_{i,l}$ and $\mathbf{W}_{l,j}$ *together*.

Dynamic Edge Pruning defines a minimum magnitude, **T**, below which any multiplication is deemed insignificant and excluded from the final computation (as shown in Equation 2).

$$\hat{\mathbf{Y}}_{i,j} = \begin{cases} 0 & \text{if } |\mathbf{X}_{i,l}\mathbf{W}_{l,j}| \leq T \\ \mathbf{X}_{i,l}\mathbf{W}_{l,j} & \text{if } |\mathbf{X}_{i,l}\mathbf{W}_{l,j}| > T \end{cases} \tag{2}$$

Fig 1 illustrates the dynamic nature of *Dynamic Edge Pruning* for linear layers and explains how it significantly reduces the error that can come with train-time unstructured pruning. Figure 2 outlines how generalized edge pruning differs from modern understandings of what techniques are possible.

With linear layers, edge pruning is already utilized. Convolutional layers, however, even when pruned in an unstructured manner, remove *correlated* edges as outlined in Figure 1. The word unstructured usually refers to the structure of the *weight tensors* as opposed to the network itself. With *dynamic edge pruning*, we can skip individual multiplications, taking advantage of the whole filter when it is useful and skipping parts of it when it's not worth doing the computation.

2.2 Location-Specific Thresholding

For successful Dynamic Edge Pruning, we propose *Location-Specific Thresholding*, which applies specialized thresholds based on the magnitude of the thresholded term to skip multiplications.

Algorithm 1: Threshold Calculation for Location-Specific Thresholding

input : A threshold for the minimum magnitude of a multiplication T and a control term C. Here, C is either weight, W or input, X.
output : The magnitude of the quotient of T and C.
Function *GenerateThreshold(T, C)*:
  return $|T/C|$
end

The most obvious threshold specialization method is to divide **T** by the absolute value of the static parameter, $\mathbf{W}_{l,j}$ as is outlined in Algorithm 1.

$$\hat{\mathbf{X}}_{l,j} = \begin{cases} 0 & \text{if } |\mathbf{X}_{i,l}| \leq \bar{\mathbf{W}}_{i,l} \\ \mathbf{X}_{i,l} & \text{if } |\mathbf{X}_{i,l}| > \bar{\mathbf{W}}_{i,l} \end{cases} \text{where, } \bar{\mathbf{W}}_{\mathrm{l,j}} = \frac{T}{|\mathbf{W}_{l,j}|} \tag{3}$$

As $\mathbf{W}_{l,j}$ is fixed, we can calculate $\hat{\mathbf{X}}_{l,j}$ during training. However, this doubles the network's memory requirement, which is often unfeasible for battery-free devices as they have extremely limited low memory (e.g., 256 KB for MSP430FR5994). Thus, we propose calculating these thresholds at runtime.

Convolutional layers are designed to exploit spatial locality in data (e.g., images), with weights organized in smaller kernels applied across the input feature map. Since each

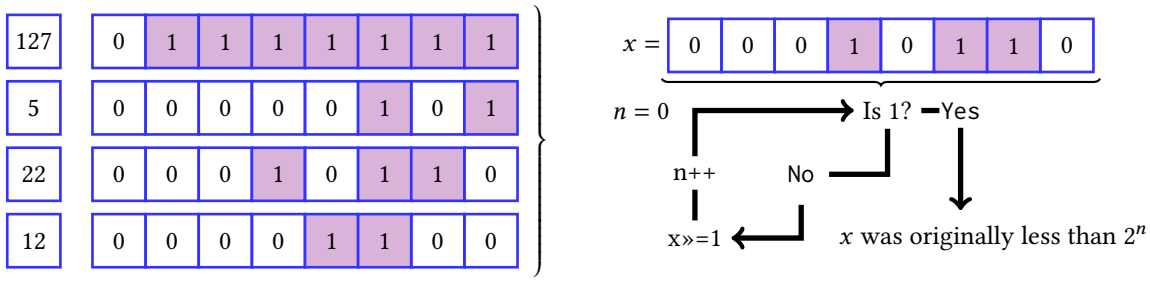


Figure 3: For any integer or fixed-point number x, by performing up to ω bit shifts to the right, we are able to determine the approximate order of magnitude of the original number where ω is the word size of the processor.

kernel has relatively few weights and each weight is utilized in many more multiplications than any of the previous layer's activation values, we prune the input matrices using equation 3.

However, for linear layers, each neuron in the previous layer is connected to every neuron in the next, resulting in large weight matrices in which each weight is only used once. For this reason, Equation 4 allows for better amortization of the division operation as linear layers perform only one multiplication per $\mathbf{W}_{l,j}$. In Figure 2, the output from the previous layer is lined up with their associated multiplications. In reality, this means that we can only keep track of one threshold at a time and apply it many times before needing to generate a new one.

$$\hat{\mathbf{W}}_{l,j} = \begin{cases} 0 & \text{if } |\mathbf{W}_{l,j}| \leq \bar{\mathbf{X}}_{i,l} \\ \mathbf{W}_{l,j} & \text{if } |\mathbf{W}_{l,j}| > \bar{\mathbf{X}}_{i,l} \end{cases} \text{where, } \bar{\mathbf{X}}_{\mathbf{l,j}} = \frac{T}{|\mathbf{X}_{i,l}|} \quad (4)$$

2.3 Fast Division Approximation

Though replacing many multiplications with relatively fewer divisions and an equivalent number of comparisons is more efficient, division remains an expensive operation to perform on any device, especially so with micro-controllers. Thus, we propose three fast division approximation techniques based on—(1) bit shifting, (2) binary tree search, and (3) bit masking. Each of these techniques provides unique benefits for different types of devices. Bit shifting and binary tree are focused on integers and fixed-point numbers and thus are suitable for extremely low-power micro-controllers that do not have floating point units (e.g., MSP430FR5994). Bit Masking operates on floating-point numbers, making it suitable for devices with floating-point units (e.g., MAX78000).

Bit shifting. For fixed-point and integer numbers, a series of bit shifts can be used to approximate division. Bit shifting is an efficient method for dividing by powers of 2, leveraging the binary structure of numbers. Instead of performing traditional division, which can be computationally expensive, our algorithm shifts the bits of the absolute value of the input to the right until it becomes 1. Doing so gives us a truncated value of the input in the form of a power of 2, as can be seen in Figure 3. This can then be used to approximate division by subtracting the exponent from the exponent of our chosen **T** value. We then perform a leftwards bit shift on 1 to generate our threshold, equal to this power of 2.

The maximum amount of time that this algorithm can take is directly proportional to the processor's *word size*. With 16-bit processors such as the MSP430[3], the cost is low relative to 32 or even 64-bit processors, making the algorithm better suited for 16-bit and even 8-bit processors. That said, the initial shift can be more than a single place or a different comparison can be made, quantizing the value even further but saving even more time. It's important to note that with fixed-point numbers, the exponent calculated will be relative to the fixed point as opposed to being absolute. This can be adjusted for by initializing the value n as shown in Figure 3 to a different value than 0.

Binary Tree. A binary tree can be utilized to quickly determine the order of magnitude of a given input. This is done by performing a series of comparisons with pre-computed pivots to determine the order of the input, as is demonstrated in Figure 4. Similar to the bit-shift method's ability to be sped up by initially shifting more or using a higher comparison, the pivots can be moved depending on the frequency of various magnitudes, making some lookups faster than others.

Unlike the previous method, this is capable of working on devices using any type of number. It's recommended, however, that it only be used for integers or fixed-point numbers whose bits have been reinterpreted as integers. This is because for floating-point numbers, the range in orders of magnitude is proportional to the maximum value that can be stored in the exponent bits, making increasing word sizes correlate to exponentially more expensive lookups than linearly more expensive ones. As with the previous algorithm, the exponent generated will be relative to the fixed point.

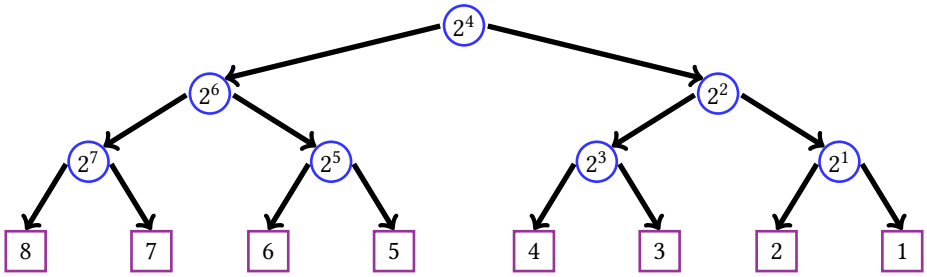
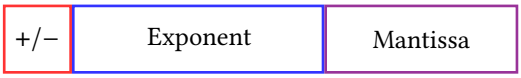


Figure 4: Starting with the head, the value at each sub-tree's head is compared to the value we are checking. If the value is less, then we descend down the tree to the right, otherwise the left. Repeating this until reaching a leaf results in the smallest value n such that 2^n is greater than the checked value.

Bit mask. The bit mask algorithm is rather unintuitive but one of the most widely applicable. This algorithm relies heavily on how IEEE 754 floating-point numbers[2] are stored, which can be seen below.



The first bit in a floating point number represents the sign. The rest represent the *exponent* and the *mantissa*. We can therefore rewrite these numbers as follows:

$$(-1)^S \left(2^{(E-E_0)}\right)\left(1+\frac{M}{M_{\text{Max}}}\right)$$

Here, S is the sign bit, E is the relative exponent, E_0 is the exponential offset, M is the mantissa, and M_{Max} is the maximum value that the mantissa can hold plus 1. This allows for a large range of numbers to be stored. For our purposes, we only care about the *exponent*. We can pull out this exponent by using a bit mask of the form 01111000[1] along with a *bitwise and*. The output is then subtracted from the exponent of **T** to generate the new exponent. This value is pushed back into the normal range by adding the exponential offset again before reinterpreting the bits as a floating point number. The intuition for this can be observed below.

$$\left|\frac{(-1)^{S_1}\left(2^{(E_1-E_0)}\right)\left(1+\frac{M_1}{M_{\text{Max}}}\right)}{(-1)^{S_2}\left(2^{(E_2-E_0)}\right)\left(1+\frac{M_2}{M_{\text{Max}}}\right)}\right| = \frac{\left(2^{(E_1-E_0)}\right)}{\left(2^{(E_2-E_0)}\right)}\frac{\left(1+\frac{M_1}{M_{\text{Max}}}\right)}{\left(1+\frac{M_2}{M_{\text{Max}}}\right)}$$

Since M must be smaller than M_{Max}, the value of the second term will be between $\frac{1}{2}$ and 2. Therefore this can be simplified to the following.

$$\frac{\left(2^{(E_1-E_0)}\right)}{\left(2^{(E_2-E_0)}\right)}\frac{\left(1+\frac{M_1}{M_{\text{Max}}}\right)}{\left(1+\frac{M_2}{M_{\text{Max}}}\right)} \approx 2^{(E_1-E_2)}$$

The conversion to a floating point number only requires adding the exponential offset back results in the desired threshold. This is only possible with floating point numbers as it takes advantage of how they are stored so heavily. Additionally, due to limitations with how C works as a programming language, implementing this requires inlining assembly because reinterpreting bits can only be done on values stored in RAM or on the stack as opposed to within registers, making the overhead far too high.

3 Experimental Setup

This section describes the evaluation setup, including platform, dataset, network, baseline, evaluation metrics, and implementation details.

3.1 Evaluation Platform

We evaluate our proposed algorithm using MSP430FR5994 [3], a common micro-controller for battery-free systems [9, 19]. MSP430FR5994 requires significantly higher multiplication than branching due to having such a limited instruction set and no pipelining. To test on the MSP430, we utilized a modified version of SONIC [11], a neural network runtime built for intermittent systems. To measure the energy consumption, we used Code Composer Studio's EnergyTrace™ Technology, which can measure the wattage of the MSP430 in real time.

We evaluate algorithms targeting floating-point operations on more standard hardware, as the MSP430FR5994 does not support floating-point arithmetic. To facilitate this evaluation, we modify PyTorch, as will be described in section 3.4.

[1]This is for 8-bit floating point numbers, but the same idea works for all sizes.

MNIST	CIFAR-10	HAR	KWS
C: $6 \times 1 \times 5 \times 5$	C: $6 \times 3 \times 5 \times 5$	C: $6 \times 1 \times 5$	C: $6 \times 1 \times 5 \times 5$
P: 2×2	P: 2×2	P: 2	P: 4×4
C: $16 \times 6 \times 5 \times 5$	C: $16 \times 6 \times 5 \times 5$	C: $16 \times 6 \times 5$	C: $16 \times 6 \times 5 \times 5$
P: 2×2	P: 2×2	P: 2	P: 4×4
L: 256×10	L: 400×10	L: 112×10	L: 288×12
C: Convolutional Layer	L: Linear Layer	P: Max Pool	

Figure 5: Model Architectures Used For Testing

3.2 Dataset

We evaluate the performance of *UnIT Pruner* across datasets spanning different domains. Specifically, we conduct experiments on an image classification dataset (MNIST [7], CIFAR-10 [22]), an acoustic dataset (Google Keyword Spotting (KWS) [41]), and a motion-based human activity recognition dataset (HAR) [18]. The training data is divided into training and validation subsets for each dataset. After shuffling, 90% of the training data is allocated to the training set, while the remaining 10% is used for validation. The test dataset is kept entirely separate and is only utilized for inference to ensure an unbiased evaluation of Sys's effectiveness.

3.3 Network

We evaluate *UnIT Pruner* on small deep neural networks specifically designed to fit into the 256KB memory of MSP430-FR5994 [3]. Figure 5 outlines the network architecture for each dataset. All the models have two convolutional layers, each followed by a max pooling layer and, finally, a linear layer. For the CIFAR10, KWS, and MNIST datasets, we use 2D convolution and 2D max pool, and for the HAR dataset, we use 1D convolution with 1D max pool. These models are trained using PyTorch before being quantized to work with the SONIC[11] runtime for the MSP430, which only works with fixed-point numbers.

3.4 Baseline

We compare *UnIT Pruner* against the unpruned version, train-time pruning, and state-of-the-art inference-time pruning, FATReLU[23]. For train-time pruning, we perform unstructured magnitude-based global pruning. Like ReLU, FATReLU is an activation function that sets values below a certain threshold to 0. It replaces the default threshold value with a higher one to intentionally increase sparsity in the network during inference. We choose FATReLU as the only existing inference-time pruning method with reasonable overhead to deploy in a battery-free system.

3.5 Evaluation metrics

We evaluate *UnIT Pruner* using four different metrics.

Accuracy Drop. We calculate the performance drop due to the proposed method's accuracy from a baseline model. We choose the unpruned model performance as our baseline model. **MACs Skipped**. We calculate how many MAC operations are skipped due to the proposed method.

Power Consumption. We calculate the power consumption during inference with and without the overheads such as data transfer.

Execution Time. Similar to the power consumption, we measure the execution time for inference, including data transfer and other overhead time.

3.6 Implementation Details

PyTorch. To support inference-time pruning, we reimplement PyTorch's linear and convolutional layers in C++[2]. By customizing these layers, we gain finer control over layer operations. It enables efficient adjustment and testing of threshold values. This modification further allows faster experimentation and tuning, as C++ provides lower-level control and typically faster execution than Python alone.

This code is available as a Python module and will include linear, 1D convolutional, and 2D convolutional layers. In addition to the *UnIT Pruner* versions of the modules, we will release a debug version. This debug version hinders performance greatly but gives statistics on how many multiplications are skipped during runtime.

C++. The PyTorch module utilizes a C++ backend that allows granular data control. Due to Python being an interpreted language, C++ bindings are used for applications where performance is of concern. In the case of PyTorch, almost all manipulation of tensors is handled by C++, making it impossible to modify PyTorch's core in the way that was necessary for these experiments without changing the underlying C++ code. To determine the efficiency of our division algorithms, we utilize C/C++ on its own without the support of Python

[2]Code available here: https://github.com/anonymouspaper2314

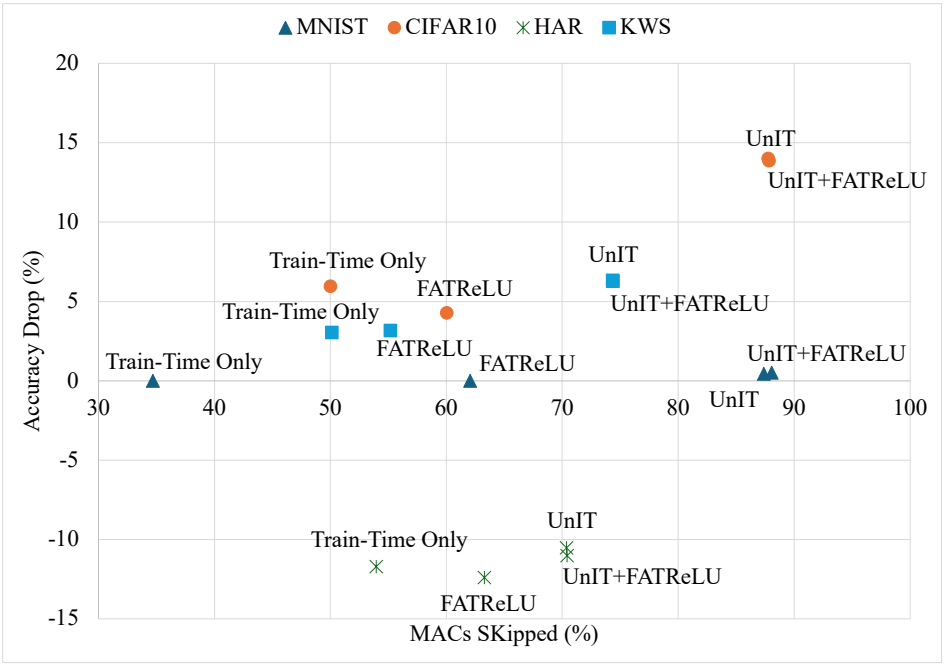


Figure 6: Performance of UnIT regarding MAC operations skipped and accuracy drop. UnIT can consistently skip a high number of MAC operations without any significant performance drop

to better understand what was happening under the covers and measure the algorithm more directly.

SONIC. We modify the SONIC[11] runtime to include support for *UnIT Pruner*. We extend their fixed-point number implementation to include our division approximation techniques. Namely, we implement the bit-shift and binary tree techniques. We modify the implementation of convolutional and linear layers to generate and utilize the thresholds as needed. We also implemented activation sparsity skipping to further demonstrate the interoperability of *UnIT Pruner* with other techniques.

4 Results

In this section, we evaluate the performance of our proposed method in terms of speed, energy, and accuracy.

4.1 Performance Speed Trade-off

Figure 6 shows the model performance with different pruning methods for four datasets: MNIST, CIFAR10, HAR, and KWS. *UnIT Pruner* skips 70.38 - 87.39% MAC operation with as low as 0.51% accuracy drop. Our highest accuracy drop (14%) was with CIFAR10, where we skipped 87.79% of the multiplications. With a small baseline model, the unpruned performance on CIFAR10 is already limited therefore a slight accuracy drop caused by *UnIT Pruner* becomes more significant. For the KWS dataset, the maximum accuracy drop is only 6%. Overall, *UnIT Pruner* skips many unnecessary multiplications without a significant accuracy drop, which is crucial for a batteryless system.

Moreover, The proposed method has 16.43–52.7% less MAC operation compared to train-time pruning with a 0.43-1.19% accuracy drop. Compared to FATReLU, our model achieves 7.08–25.34% less MAC operation with 0.43–1.88% less accuracy. Thus, our method can be more efficient with less MAC operation by sacrificing a small performance.

UnIT Pruner not only achieves comparable performance when skipping the vast majority of MAC operations but is also a more efficient pruning method for batteryless systems than FATReLU. Figure 6 shows that our method not only outperforms FATReLU but integrating FATReLU with *UnIT Pruner* does not give any advantage, making our method the most effective in these tests.

4.2 Time Analysis

Time spent moving memory around was a *major* contributor to the total runtime of our tests. Therefore, we subtract the time it takes to do everything *except* MAC operations from all final times. Figure 7 shows that *UnIT Pruner* drastically reduces inference time by 78.8–94.7% compared to train-time pruning and 74.4–89% faster than FATReLU, achieving times of 7.5–151 and 3.8–125 seconds, respectively. We also observe that *UnIT Pruner* has a lower time overhead than FATReLU.

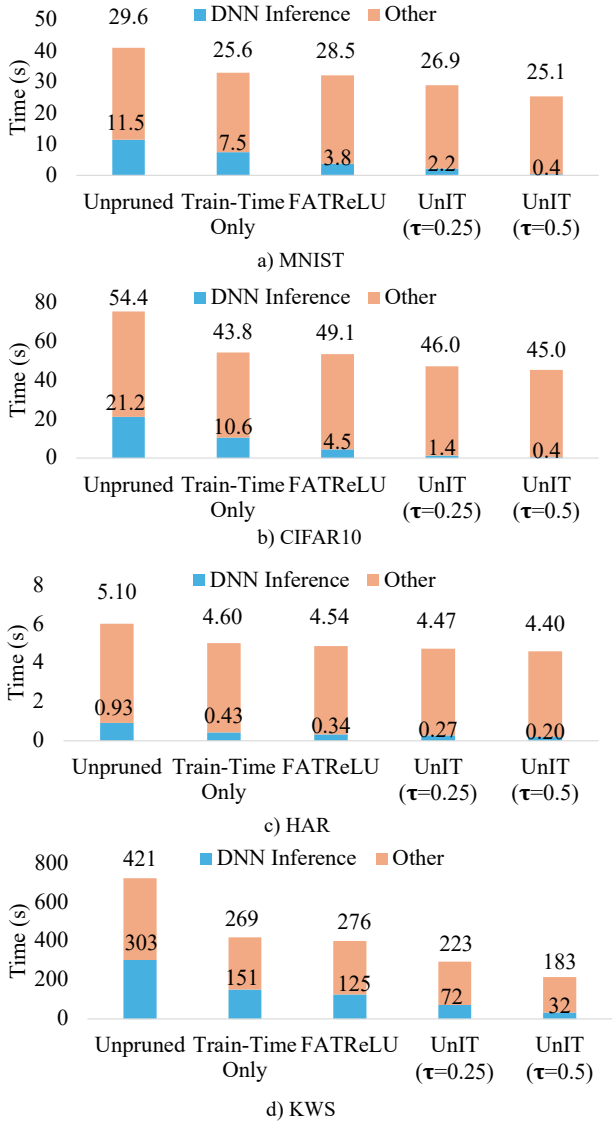


Figure 7: UnIT achieves faster inference than other pruning methods for all four datasets.

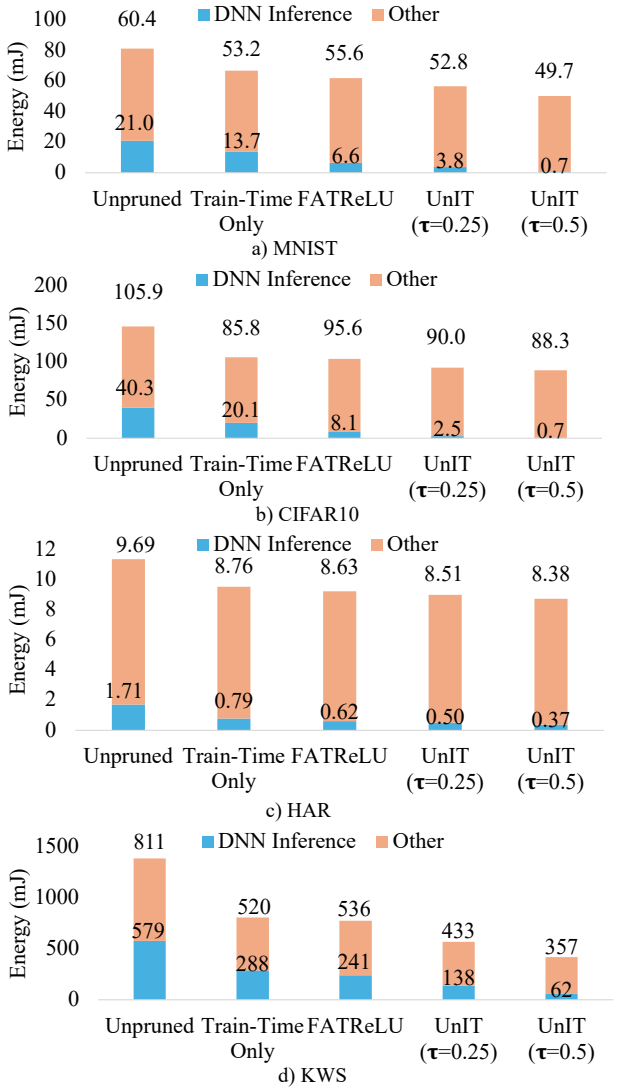


Figure 8: UnIT consumes less energy than other pruning methods for all four datasets.

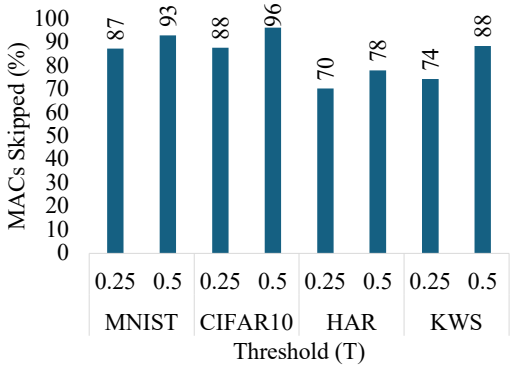
As expected for a battery-free system, most of the time is spent moving data and other computations.

4.3 Energy Analysis

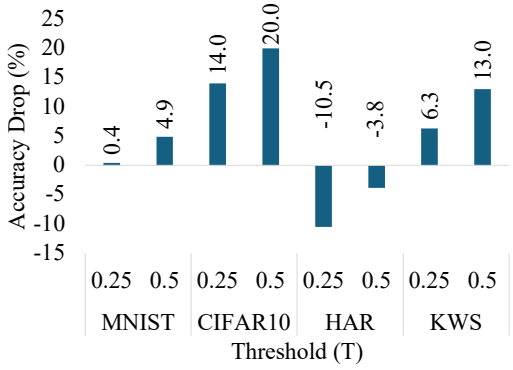
In Figure 8, UnIT consumes only 0.37–62mJ per inference on the MSP430, whereas FATReLU needs 0.62–241mJ, and train-time pruning needs 0.79–288mJ. Even with the energy for data transfer, overhead, and other computational tasks, our algorithm saves the most energy when compared to other methods, making it suitable for battery-free systems.

4.4 Impact of Threshold

Figure 9 demonstrates the impact of threshold choice on UnIT's performance. With a lower threshold, we skip fewer MAC operations while maintaining a higher performance. The higher the threshold, the more significant the performance drop. For this work, we set the threshold to 0.25 for all results. However, even with a low threshold, our method achieves the best performance of the tested pruning methods, as is evident from previous sections.

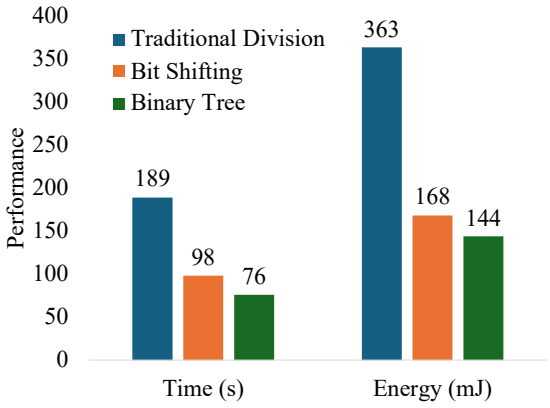


a) MACs skipped

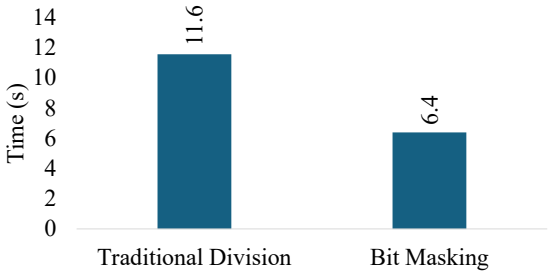


b) Accuracy drop

Figure 9: Increasing the threshold skips more MAC operations and lowers accuracy.



a) Bit shifting and binary tree on MASP430



b) Bit Masking on Intel core i7

Figure 10: Fast approximate division using bit shifting, binary tree search, and bit masking outperforms traditional division techniques with significantly faster and cheaper operations.

4.5 Fast Division Approximation Techniques

We compare the performance of our fast division approximation techniques with that of traditional divisions. We perform bit shifting and binary tree searches on the MSP430 as they are suitable for integer and fixed point systems. Although we cannot compare the performance of the bit masking method with the traditional division on the MSP430 due to the unavailability of a floating point system, we perform bit masking on an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s), 12 Logical Processor(s) for 10 billion iterations. Figure 10 shows the performance of our fast approximate division techniques compared to the traditional division. Bit shifting and binary tree search both achieve 50-59.8% lower execution time and 53.7 - 60.3% lower energy consumption when compared to the conventional division method on the MSP430. Similarly, bit masking completes 44.8% faster than the traditional method.

5 Related Work

DNN in Battery-free Systems. DNN in battery-free systems requires additional processing to make the model memory and computationally efficient. Researchers have deployed train-time pruning [29], special pruning [14], and quantization [43] to make the model smaller. Tensor decomposition and early-exit strategies have been popular in reducing inference time for on-device computation [9, 19, 34, 39]. Neural architecture search (NAS) is used to find the most compressed model to run on battery-free systems [11, 31]. Some systems utilize Knowledge distillation to make a smaller model from a larger model [5, 36]. Researchers also utilized a combination of quantization or pruning with knowledge distillation for a more efficient model [37].

Structured Pruning. Structured pruning targets the removal of entire structures, such as neurons, channels, or even layers. This approach simplifies model architecture in a way that aligns well with hardware, as it results in smaller matrices or tensor dimensions. Notable structured pruning

methods include *channel pruning* [25], which removes channels based on a criterion like weight magnitude or contribution to overall model accuracy. *Filter pruning* [30] is another structured technique that removes filters based on the importance of their activations, reducing convolutional layer sizes in CNNs and enhancing deployment speed on hardware that benefits from reduced channel dimensions. More recent structured pruning techniques also involve *group sparsity regularization* [42], where specific regularizers are applied to groups of weights to force sparsity within predefined structures, yielding efficient models without major loss in accuracy. Structured pruning is widely favored for deployment on accelerators, as it produces compact models without the need for additional computation to handle irregular sparsity.

Unstructured Pruning. Unstructured pruning, by contrast, removes individual weights based on specific criteria, such as magnitude. The *magnitude-based pruning* approach [15] is one of the most widely used methods in this category, where weights with the smallest absolute values are removed iteratively, yielding a sparse weight matrix. This technique, while effective in significantly reducing model parameters, results in irregular sparsity patterns that are less efficient to compute on standard hardware due to the need for specialized libraries and frameworks to leverage the sparsity. Techniques such as *variational dropout* [33] use Bayesian methods to apply pruning during training, automatically determining which weights can be removed by learning a probability distribution over them. Although unstructured pruning can yield highly sparse networks, the irregularity of the resulting weights often limits practical speedup benefits during deployment.

Training-Time Pruning. Training-time pruning, also known as *dynamic pruning*, integrates pruning operations within the training process. This category includes *Gradual Magnitude Pruning* (GMP) [44], where weights are pruned iteratively throughout the training process, allowing the model to adapt and learn with progressively fewer parameters. Similarly, the *Lottery Ticket Hypothesis* [10] suggests identifying subnetworks with randomly initialized parameters that can be retrained to achieve near-original accuracy, leading to a more efficient training process. Training-time pruning can also be applied through *regularization-based techniques* [28], where sparsity-inducing norms (e.g., L_1 norm) are added to the loss function, encouraging the model to develop sparse weight patterns inherently. By pruning during training, these approaches enable the model to adapt to lower parameter counts while preserving accuracy, though they may add complexity to the training process.

Post-Training Pruning. *Post-training pruning* is applied to pre-trained models allowing practitioners to achieve model size reductions without altering the original training pipeline. In this category, *post-training quantization and pruning* [17] is a widely used method, where redundant parameters are pruned after training and then quantized to reduce memory footprint, resulting in efficient deployment. *Layer-wise pruning* [27] is another post-training approach that uses sensitivity analysis on each layer of a trained network to prune layers with minimal impact on model performance. Techniques such as *fine-tuning after pruning* [15] are commonly employed to recover any accuracy loss following pruning, where the model is slightly retrained after pruning to regain accuracy lost due to the reduction in parameters.

Inference-Time Pruning. Inference-time pruning refers to techniques that are done *during* inference. Techniques like FATReLU [23], also known as as Truncated Rectified [21], which induce more sparsity only during runtime are examples of inference-time neuron pruning. There has also been work involving inference-time channel pruning [26] where a simpler model was trained in tandem with the main model to signal when to skip certain filters. These methods are all structured, making *UnIT Pruner* the first of its kind as an unstructured inference-time pruning system.

6 Discussion and Limitations

This section discusses the shortcomings and future potential of *UnIT Pruner*.

6.1 Trade-Off Between Adaptability and Network Size

One notable advantage of *UnIT Pruner* is its ability to retain activations and weights for potential future use, offering greater adaptability compared to traditional train-time and structured pruning methods. This flexibility is especially beneficial in environments where computational and energy resources fluctuate. However, preserving these weights limits the ability to reduce the network size further. This presents a challenge, particularly given the algorithm's primary focus on resource-constrained hardware, where minimizing network size is crucial for optimal performance.

6.2 *UnIT Pruner* on Highly Parallelized Hardware

GPUs and other highly parallelized hardware may struggle to leverage *UnIT Pruner* fully. In such systems, the potential savings are diminished because all cores must wait for the slowest one, causing the worst-performing core to dictate the speed of the operation. This blocking behavior reduces the effectiveness of the algorithm on these devices. While the results presented here suggest that the frequency of skipped

operations could be high enough for certain systems to benefit, it is evident that as the number of cores increases, the overall savings will decrease. Future experiments will be conducted to verify this hypothesis.

6.3 Portability Limitations Due to Hardware-Specific Division Techniques

Proposed fast division approximation techniques significantly reduce computational overhead, making the algorithm suitable for many embedded systems. However, the need for different algorithms depending on the hardware used means that the portability of *UnIT Pruner* runtimes would be limited. This hardware dependency may restrict the algorithm's versatility across diverse platforms and complicate its deployment in varied environments.

6.4 Impact of Model Size

Since the algorithm was designed with low-power systems in mind, the testing so far has been focused on smaller models to ensure efficient resource utilization. However, the performance and benefits of *UnIT Pruner* on larger, more complex models remain untested. In future work, we plan to evaluate *UnIT Pruner* with larger models to better understand its scalability, performance under higher computational loads, and ability to maintain efficiency across a broader range of applications. This will provide a more comprehensive assessment of the algorithm's full potential and help identify any additional challenges that may arise with increased model complexity.

6.5 Expand to Diverse DNN Layers

Thus far, we have only tested *UnIT Pruner* with convolutional and linear layers, primarily due to their popularity and efficiency, especially in resource-constrained applications. Future work will include testing a broader range of layer types, such as recurrent and attention layers, to assess Sys's adaptability and performance across different architectures. This will help evaluate its general applicability to a broader set of neural network models.

6.6 Testing on Diverse Edge Devices

While the MSP430 is a good starting point, we aim to test *UnIT Pruner* on a wider range of devices, particularly edge computing devices with unique properties. We believe the flexibility of the techniques described here will be instrumental in porting the algorithm to various embedded systems, enabling its use in different hardware environments. This will help assess the adaptability and performance of *UnIT Pruner* across a broader spectrum of edge devices, contributing to its potential for more widespread deployment

6.7 Data Transfer Overhead

Although our method reduces inference time significantly, the energy and time spent moving memory on the device currently outweigh the time spent on computations, resulting in high total inference time. We plan to explore alternative, application-specific memory management systems tailored for ultra-low-power devices. Future work could investigate more efficient storage and retrieval mechanisms that better support dynamic pruning while maintaining the benefits of adaptability. Additionally, we aim to develop techniques to minimize memory transfer overhead, further optimizing the overall performance of *UnIT Pruner* on resource-constrained hardware.

6.8 Imprecise BLAS

Fundamentally, the techniques outlined here for quickly generating thresholds and applying them induce and take advantage of sparsity in matrix multiplications. While these threshold generation techniques may not be as valuable in machine learning contexts for systems with ample storage, they can still be useful in scenarios like Basic Linear Algebra Subprograms (BLAS), where the two matrix values are entirely unknown. In such cases, these algorithms may offer a form of dynamic thresholding, helping to improve efficiency by selectively skipping redundant operations. This approach highlights the potential applicability of these methods beyond traditional machine learning tasks, particularly in resource-constrained environments.

7 Conclusion

This paper presents *UnIT Pruner*, the first unstructured inference-time pruning algorithm that implements dynamic edge pruning, a novel form of pruning that is unstructured relative to the model as opposed to just the weights. By dynamically pruning individual connections based on input relevance, *UnIT Pruner* significantly reduces MAC operations, inference time, and energy consumption while maintaining competitive accuracy levels. The algorithm's adaptability and fast division approximation techniques highlight its suitability for embedded and intermittent systems.

Experimental results validate the effectiveness of *UnIT Pruner*, showing reductions of 70.38–87.39% in MAC operations, 74.4–94.7% in inference time, and 74.2–96.5% in energy consumption with accuracy drops as low as 0.43%. Future work will focus on extending *UnIT Pruner*'s capabilities to larger models and exploring its ability to be integrated with other pruning methods to further optimize performance across a diverse set of applications.

References

[1] 2018. Efficient Multiplication and Division Using MSP430™ MCUs. https://www.ti.com/lit/an/slaa329a/slaa329a.pdf

[2] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[3] 2024. MSP430 User Guide. https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf

[4] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 3 (2017), 1–18.

[5] Manoj Bharadhwaj, Gitakrishnan Ramadurai, and Balaraman Ravindran. 2022. Detecting vehicles on the edge: Knowledge distillation to improve performance in heterogeneous road traffic. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3192–3198.

[6] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. 2022. Inference-aware convolutional neural network pruning. *Future Generation Computer Systems* 135 (2022), 44–56.

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

[8] Maha S Diab and Esther Rodriguez-Villegas. 2022. Embedded machine learning using microcontrollers in wearable and ambulatory systems for health and care applications: A review. *IEEE Access* 10 (2022), 98450–98474.

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

[10] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

[11] 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.

[12] Maria Gorlatova, Peter Kinget, Ioannis Kymissis, Dan Rubenstein, Xiaodong Wang, and Gil Zussman. 2010. Energy harvesting active networked tags (EnHANTs) for ubiquitous object networking. *IEEE Wireless Communications* 17, 6 (2010), 18–25.

[13] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[14] 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).

[15] 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).

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

[17] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in neural information processing systems*. 4107–4115.

[18] Andrey Ignatov. [n. d.]. HAR. https://github.com/aiff22/HAR

[19] Bashima Islam and Shahriar Nirjon. 2019. Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems. *arXiv preprint arXiv:1905.03854* (2019).

[20] Aman Kansal and Mani B Srivastava. 2003. An environmental energy harvesting framework for sensor networks. In *Proceedings of the 2003 international symposium on Low power electronics and design*. 481–486.

[21] Kishore Konda, Roland Memisevic, and David Krueger. 2015. Zero-bias autoencoders and the benefits of co-adapting features. arXiv:1402.3337 [stat.ML] https://arxiv.org/abs/1402.3337

[22] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).

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

[24] Se Jung Kwon, Dongsoo Lee, Byeongwook Kim, Parichay Kapoor, Baeseong Park, and Gu-Yeon Wei. 2020. Structured compression by weight encryption for unstructured pruning and quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1909–1918.

[25] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2017).

[26] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/a51fb975227d6640e4fe47854476d133-Paper.pdf

[27] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2019. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8060–8068.

[28] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2017. Rethinking the value of network pruning. In *International Conference on Learning Representations*.

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

[30] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*. 5058–5066.

[31] Bo Lyu, Hang Yuan, Longfei Lu, and Yunye Zhang. 2021. Resource-constrained neural architecture search on edge devices. *IEEE Transactions on Network Science and Engineering* 9, 1 (2021), 134–142.

[32] 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.

[33] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. 2017. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2498–2507.

[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] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH*

computer architecture news 45, 2 (2017), 27–40.

[36] Peihan Qi, Xiaoyu Zhou, Yuanlei Ding, Zhengyu Zhang, Shilian Zheng, and Zan Li. 2022. Fedbkd: Heterogenous federated learning via bidirectional knowledge distillation for modulation classification in iot-edge system. *IEEE Journal of Selected Topics in Signal Processing* 17, 1 (2022), 189–204.

[37] Xiaoyang Qu, Jianzong Wang, and Jing Xiao. 2020. Quantization and knowledge distillation for efficient federated learning on edge devices. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 967–972.

[38] Maying Shen, Pavlo Molchanov, Hongxu Yin, and Jose M Alvarez. 2022. When to prune? a policy towards early structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12247–12256.

[39] 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.

[40] Kleanthis C Thramboulidis, G Doukas, and G Koumoutsos. 2007. A SOA-based embedded systems development environment for industrial automation. *EURASIP Journal on Embedded Systems* 2008 (2007), 1–15.

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

[42] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*. 2074–2082.

[43] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4820–4828.

[44] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).

Received 14 November 2024