

# Tin-Tin: Towards Tiny Learning on Tiny Devices with Integer-based Neural Network Training

Yi Hu  
Carnegie Mellon University

Jinhang Zuo  
City University of Hong Kong

Eddie Zhang  
Carnegie Mellon University

Bob Iannucci  
Carnegie Mellon University

Carlee Joe-Wong  
Carnegie Mellon University

## ABSTRACT

Recent advancements in machine learning (ML) have enabled its deployment on resource-constrained edge devices, fostering innovative applications such as intelligent environmental sensing. However, these devices, particularly microcontrollers (MCUs), face substantial challenges due to limited memory, computing capabilities, and the absence of dedicated floating-point units (FPUs). These constraints hinder the deployment of complex ML models, especially those requiring lifelong learning capabilities. To address these challenges, we propose Tin-Tin, an integer-based on-device training framework designed specifically for low-power MCUs. Tin-Tin introduces novel integer rescaling techniques to efficiently manage dynamic ranges and facilitate efficient weight updates using integer data types. Unlike existing methods optimized for devices with FPUs, GPUs or FPGAs, Tin-Tin addresses the unique demands of tiny MCUs, prioritizing energy efficiency and optimized memory utilization. We validate the effectiveness of Tin-Tin through end-to-end application examples on real-world tiny devices, demonstrating its potential to support energy-efficient and sustainable ML applications on edge platforms.

## CCS CONCEPTS

- Computing methodologies → Neural networks;
- Computer systems organization → Sensor networks.

## KEYWORDS

IoT, edge computing, on-device training

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*Conference'17, July 2017, Washington, DC, USA*

© 2025 Association for Computing Machinery.  
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00  
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## ACM Reference Format:

Yi Hu, Jinhang Zuo, Eddie Zhang, Bob Iannucci, and Carlee Joe-Wong. 2025. Tin-Tin: Towards Tiny Learning on Tiny Devices with Integer-based Neural Network Training. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

The rapid development of the Internet of Things (IoT) and edge computing has significantly expanded the deployment of machine learning (ML) on edge devices, thereby enhancing the efficiency and responsiveness of applications like wearable sensing [31, 44, 50] and smart cities [1, 12, 34]. Moreover, many emerging applications, such as wildfire monitoring [8, 43] or detecting faults in manufacturing equipment [56, 68], necessitate maintenance-free, long-lived sensor networks [26, 48, 58] that can deploy ML models to detect anomalies and make predictions about the environment. In these scenarios, low-power sensors and microcontrollers (MCUs) play a critical role, being both cost-effective and disposable, to ensure sustainability and longevity.

However, enabling ML on these resource-constrained devices presents many new challenges. Power-efficient MCUs typically have limited memory and computing capabilities, as detailed in Sec. 3.1. For instance, the popular STM32 Cortex M0<sup>1</sup> MCUs have up to 144 KB of data RAM, and the MSP430<sup>2</sup> mixed-signal MCUs offer up to 66 KB. In addition to memory constraints, both lack dedicated floating point units (FPUs), which are designed to handle operations on floating point numbers. Consequently, these MCUs can only perform integer arithmetic or rely on slow software implementations for floating-point calculations. These limitations significantly restrict the complexity of ML models that can be deployed on these edge devices.

---

<sup>1</sup><https://www.st.com/en/microcontrollers-microprocessors/stm32f0-series.html>

<sup>2</sup><https://www.ti.com/microcontrollers-mcus-processors/msp430-microcontrollers/overview.html>

Furthermore, most existing research on edge ML focuses only on inference tasks. However, in practical edge applications with evolving environments, on-device lifelong learning [35, 76] becomes crucial. Devices must dynamically learn new tasks and continuously adapt to evolving input distributions. In manufacturing applications, for example, it is common for real operating conditions to differ from test ones [11, 46], requiring a fault detection model to be re-trained in real time on operational data. Moreover, manufacturing equipment degrades, often in an unpredictable manner, over time; sensors using ML to predict faults, then, must adapt their models to this degradation over time [17, 72]. Training, unlike inference, imposes significantly higher demands on memory and computational resources (Sec. 3.1, 3.2). For instance, it may require 7× longer compared to inference as demonstrated in [69]. This challenge is particularly demanding for tiny edge devices like MCUs, where battery life is a critical concern, necessitating efficient resource management to prevent rapid battery drain during training.

Our goal is to support and optimize *tiny, lifelong* data learning on these *highly constrained* low-power MCUs with only hundreds of KB memory while extending battery life. Efforts like TinyML [3, 33] and model compression techniques [5, 23, 40, 62] to reduce the energy usage and computational overhead of on-device model deployment are effective for inference but cannot handle model updates. Meanwhile, model training approaches using reduced-precision models and weight updates [6, 15, 37, 75] still rely on intensive intermediate floating point computations and may require specialized non-standard hardware. To the best of our knowledge, *ours is the first integer-based on-device training framework designed specifically for low-power MCUs*. While NITI [63] addresses some basic challenges of integer-based training, it is primarily tailored for GPUs and FPGAs rather than tiny MCUs, leaving several critical questions unanswered for deployments on power-constrained devices, including the two technical challenges that we address below. Table 1 offers a comprehensive comparison of our approach to related work.

To enhance energy efficiency in lifelong learning while satisfying memory and computing constraints, we introduce **Tin-Tin**, an innovative integer-based on-device training framework for **Tiny** learning on **Tiny** MCUs. Specifically, our approach replaces full-precision floating points with reduced-precision low-bitwidth data types, significantly decreasing memory requirements by using smaller data formats (e.g., fp32 to int8) and simplifying calculations (e.g., floating point to integer arithmetic). This reduction complements existing structure- and network-level optimizations like sparse updates [42] and pruning [47]. Unlike existing methods optimized for IoT devices with FPUs [42] or GPUs

and FPGAs [63], Tin-Tin specifically targets the unique demands of tiny MCUs by prioritizing energy efficiency and optimized memory utilization, effectively addressing two primary **technical challenges**:

*Restricted Representation Range*. Using integer arithmetic for neural network training requires integer representations of weights ( $w$ ), activations ( $a$ ), gradients ( $g$ ) and errors ( $e$ ). Unlike floating points, which have a wide representation range (e.g.,  $1.4e^{-45}$  to  $3.4e^{38}$  for fp32), integers have a limited range (e.g.,  $[-128, 127]$  for int8). Fixed ranges based on presumed value limits can lead to overflow (unwanted clipping) or reduced precision (Sec. 3.3). To maintain accuracy, value ranges should be dynamically adjusted during training, as the distributions of inputs, outputs, and weights change. This dynamic adjustment could enhance model performance but necessitates an appropriate rescaling scheme throughout the training process, an aspect unaddressed in previous integer-based training methods.

*Integer Weight Update*. Maintaining accurate model updates with integer weights is challenging due to their limited precision. The desired update values often need to be much smaller than the weights themselves. Traditional gradient descent techniques achieve this by using a small learning rate (e.g., 0.001) to scale down the gradient update, guiding the model towards the optimum. However, in integer-based training, directly applying a small learning rate is problematic because multiplying integer gradients by a small scalar produces floating-point values. Converting these scaled floating-point gradients back to integers introduces additional computational costs and precision issues. Therefore, innovative methods are required to apply integer gradient updates that are much smaller in scale than the weights, while maintaining accuracy within the integer-only framework.

To tackle these challenges and target resource-constrained MCUs, we introduce novel integer rescaling techniques that support a dynamic range and efficient weight updates. We demonstrate the effectiveness of our framework through comprehensive end-to-end application examples on tiny devices. Our primary **contributions** include:

- We propose Tin-Tin, a pioneering framework tailored for on-device training on MCUs, leveraging integer-based model quantization to significantly reduce memory requirements and improve energy efficiency.
- We introduce an integer-only rescaling method and a compound exponentiation scaling scheme to provide dynamic range support, ensuring accurate scale management throughout training and minimizing overflow and quantization errors. By dynamically adjusting scaling factors, Tin-Tin maintains training stability and performance on MCUs.

- We develop an efficient weight update mechanism that aligns gradients and weights to a common scale through integer-based operations, including gradient alignment and adaptive update step sizes, preserving model accuracy and efficiency during training.
- We evaluate Tin-Tin across various learning applications, focusing on its energy and memory efficiency on different MCUs. Case studies on motor bearing fault detection and spectrum sensing demonstrate Tin-Tin’s ability to significantly reduce training time, memory usage, and energy consumption. Our real-world experiments on MCUs validate that Tin-Tin can save up to 82% in energy and 40% in memory compared to full-precision floating-point implementations.

The remainder of the paper is organized as follows: Section 2 contrasts our work with related studies. Section 3 provides the background and motivation for our research. Section 4 introduces the Tin-Tin framework and discusses our novel integer rescaling techniques for managing dynamic ranges and facilitating efficient weight updates. Section 5 presents the experimental validation of our work on various MCUs. Finally, we conclude in Section 6.

## 2 RELATED WORK

*Model Compression & Optimization.* Many methods have been developed to fit large ML models for deployment on small devices with limited memory, as surveyed by Liang et al. [39]. Techniques include pruning [23–25, 38, 45, 47], quantization [4, 22, 62, 74], and neural architecture search [10, 19, 32, 40, 41, 60]. Some methods integrate these techniques during training [2, 66], such as quantization-aware training [30], which simulates quantization effects, and IQN [73], which incrementally quantizes models through re-training. While these methods effectively reduce model size and achieve faster inference, they do not address the memory constraints during training. In contrast, our work focuses on continual on-device learning.

*Memory-saving Techniques.* Efficient on-device training under memory constraints has been widely explored for DNNs. Some methods reduce memory usage by recomputing some results on-the-fly rather than saving them [13, 20, 52, 64], which incurs significant computation overhead and exacerbates on-device training time. Other works [9, 28, 36, 42, 54, 55, 65] explore sparse updates that select only a subset of layers (and tensors) to update during backpropagation to reduce memory consumption and computation. However, they require non-trivial training, such as the tiny training engine [42] and meta-training [36], or extensive analysis to identify important layers and channels relevant to the target data (e.g., contribution analysis [42], pruning [54]). This process may not be feasible if the training dataset is

unavailable, e.g., if it is confined to MCUs and cannot be easily moved to more powerful devices for analysis, and adapting to new data distributions after deployment is challenging. Additionally, these techniques mainly target deep networks and involve extensive floating-point operations. In contrast, our work focuses on simple learning tasks common on battery-powered sensor devices dealing with time-series data (e.g., temperature, acceleration) and aims to minimize floating-point operations.

*Reduced-precision Training.* The use of low bit-width reduced-precision data types for model parameters has received significant attention for its potential to reduce the computation cost (e.g., binary [16, 29], ternary [37, 75], XNOR [57], WAGE [67]). However, these solutions remain impractical for low-power sensor devices because heavy intermediate floating-point calculations are still required at certain computation stages, which can be highly inefficient on tiny devices. For instance, WAGE [67] accumulates weights with high precision in fp32 before constraining to 8-bit integers. Some of them rely on non-standard number systems (e.g., binary, nonuniform [53]) that requires dedicated hardware support. Several related works address the determination of quantization scales for reduced-precision training. PACT [14] optimally determines the quantization scale for activations, LSQ [18] and LQ-Nets [71] optimize the quantizer for weights, while LSQ+ [7] extends LSQ to asymmetric quantization. These approaches apply to full-precision models.

NITI [63], an integer-only training framework for DNNs, is the closest work to ours. NITI addresses the dynamic range issue through an innovative exponentiation scaling scheme for integer values. Unlike NITI, which fixes the weight scales, our integer-based rescaling scheme allows weight values to be dynamically adjustable during the entire training process, providing additional flexibility and precision. Moreover, NITI focuses on GPU and FPGA deployments, while we design Tin-Tin for end-to-end MCU sensing applications.

## 3 BACKGROUND & MOTIVATION

Our work is motivated by two key factors. First, integer-based model quantization can substantially reduce device memory requirements. Second, floating-point operations consume a considerable amount of energy. We then demonstrate that model weights can vary significantly during training, highlighting the necessity for a dynamic weight range.

### 3.1 Device Memory Constraints

Low-power MCUs typically have only hundreds of KB of memory (data RAM) or less to maintain low-power, low-budget operations. Figure 1 shows the RAM sizes of some popular low-power devices from EEMBC ULPMark®, an

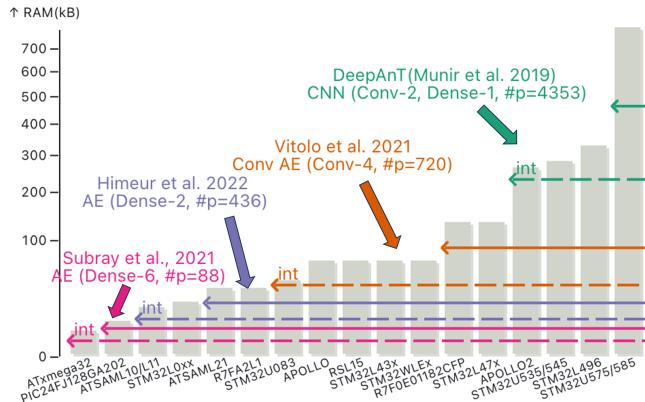
**Table 1: Related Work and Comparison.**

	Optimization Target	Applied Stage	Pre-runtime Data Not Required	Integer Backprop	Dynamic Weight Range	Low-Power MCU Deployment <sup>†</sup>
Model compression & optimization [19, 22, 23, 30, 41]	network, layers, connections	post-training, training*	no	no	N/A	M7 [41]
Sparse update [28, 36, 42, 54, 65]	backward layers, connections	pre-runtime, runtime	no	[65]	N/A	M7 [42] M4 [54]
Reduced-precision training [37, 75]	arithmetic, data	runtime	yes/no	no	no	N/A
NITI [63]	arithmetic, data	runtime	yes	yes	no	N/A
Tin-Tin ( <i>this work</i> )	arithmetic, data	runtime	yes	yes	yes	M0, M4

\* The term “training” is used separate from “runtime” to emphasize the clear distinction between model training phase and the inference phase in these works.

<sup>†</sup> “M” refers to the Cortex M series.

## ULP MCU Memory Constraints



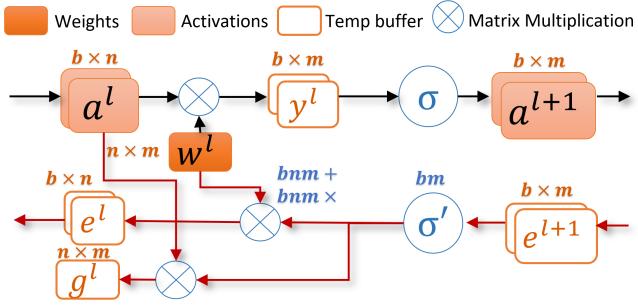
**Figure 1: Data RAM of ULP MCUs and the memory requirements for various ML applications, including anomaly detection (Himeur et al. [27], Munir et al. [49], Vitolo et al. [61]) and signal classification (Subray et al. [59]). Solid lines: fp32; dashed lines: int8. For DeepAnT [49], a kernel size of 3 and a pooling layer that preserves the output dimensions are assumed. Integer-based ML frameworks significantly reduce RAM requirements, expanding the range of devices on which ML applications can be deployed.**

ultra-low power MCU benchmark that quantifies the low-power behavior of MCUs<sup>3</sup>.

<sup>3</sup><https://www.eembc.org/ulpmark/>

Limited memory budgets significantly constrain ML applications on tiny devices. For both inference and training, the model needs to be loaded into memory. Training requires substantially more memory than inference due to the additional memory needed for backpropagation. In inference-only scenarios, all parameters of the pre-trained model can be accessed in read-only mode, and intermediate activations between layers can be discarded as data flows to the next layer. However, during training, activations must be stored to calculate gradients during the backward pass, further exacerbating memory consumption.

In general, MCUs are not designed for complex computations, such as deep vision tasks, without significant hardware and energy support. Therefore, targeted tiny learning applications should be lightweight and simple, benefiting autonomous and continual monitoring without heavily burdening battery life. One motivating application is anomaly detection, achievable via a simple auto-encoder (AE). Figure 1 shows the model architecture and the number of parameters (#p) used in four works for ML-based anomaly detection [27, 49, 59, 61]. As shown in Figure 2, we estimate the memory usage considering three types of memory consumption for training: (1) loading all model parameters, (2) storing activations during the forward pass, and (3) dynamically allocated memory during the process (e.g., temporary buffers for storing gradient results). The first two are typically statically allocated on MCUs, while the third is estimated as the sum of the sizes of the largest weight and activation in full precision (fp32/int32) across layers for storing intermediate activation and gradient results. The estimated memory usage of using fp32 and int8 for storing model parameters



**Figure 2: Dataflow illustrating the memory requirements and computation costs of the forward (black) and backward (red) passes for a linear layer with input dimension  $n$ , output dimension  $m$  and batch size  $b$ .**

**Table 2: Measured latency, power and energy consumption for a single matrix multiplication involving two  $8 \times 8$  matrices. Integer (int8) computations consume significantly less energy compared to floating point (fp32) on all devices.**

Device	Freq (Hz)	t ( $\mu$ s)	P (mW)	E ( $\mu$ J)
Feather M0 ATSAMD21	48M	fp32	3550.86	44.15
		int8	<b>148.06</b>	<b>5.28</b>
Feather RP2040	125M	fp32	641.37	102.94
		int8	<b>74.32</b>	<b>8.15</b>
expLoRaBLE Apollo3/FPU	96M	fp32	133.45	13.29
		int8	<b>100.68</b>	<b>1.35</b>

and activations with a batch size of 32 is shown in Figure 1. Using integer weights and parameters can mitigate memory constraints and enable learning on more low-power devices with as little as 2 KB of memory (e.g., ATxmega32).

### 3.2 Expensive Floating Point Operations

Training a neural network requires significant computational resources, mainly for matrix multiplication. Figure 2 shows the computation involved in the forward and backward passes of a linear layer  $l$ , which includes one matrix multiplication in the forward pass and two matrix multiplications in the backward pass. With input dimension  $n$ , output dimension  $m$ , and batch size  $b$ , each matrix multiplication requires  $bnm$  multiply-add operations (MACs), totaling  $2bnm$  floating point operations (FLOPs), in addition to activation functions on each output that require  $bm$  FLOPs per pass. The calculations for convolutional layers can be more complex, with the number of matrix multiplications increasing with the number of channels and the dimension of the output.

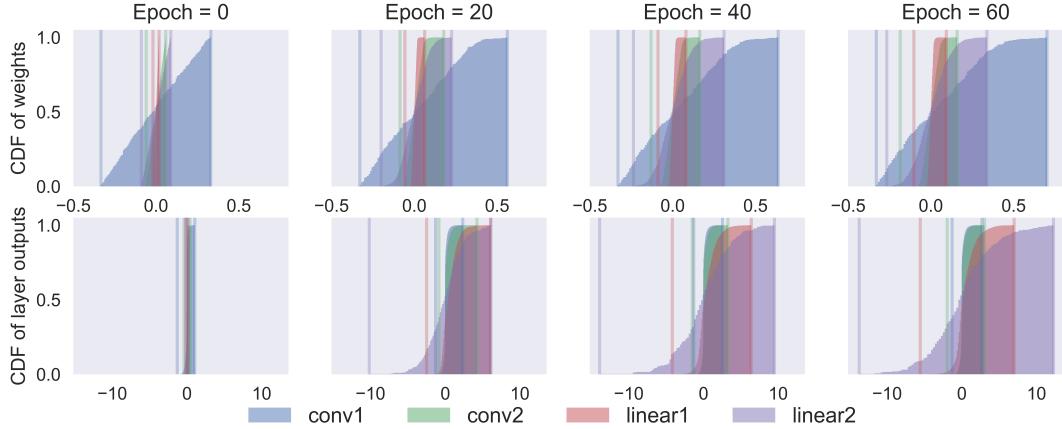
Performing matrix multiplications with floating points is very expensive in terms of both latency and energy, especially on MCUs without FPUs, which must rely on slower software implementations. We measured the latency of a single matrix multiplication of two  $8 \times 8$  matrices (using three nested for loops) on three MCU boards with different computational capabilities. A power analyzer was used to measure the current consumption. The results of using floating point data type (fp32  $\times$  fp32  $\rightarrow$  fp32) and integer data type (int8  $\times$  int8  $\rightarrow$  int32) are summarized in Table 2. Devices without FPUs (e.g., ATSAMD21, RP2040) are very slow on FLOPs, increasing energy consumption by up to 30 times. Therefore, replacing floating points with integers can significantly improve energy efficiency on these tiny devices with limited compute capabilities. Integer calculations are also faster on more powerful edge devices. For example, the same matrix multiplication takes 228.37  $\mu$ s with floating points and 203.05  $\mu$ s with integers on a laptop (MSI Summit E13 Flip Evo), and 196.86  $\mu$ s with floating points and 170.99  $\mu$ s with integers on an iPhone 15. Although many prior works have focused on quantizing neural networks to improve latency and energy efficiency, *we are the first to incorporate integer calculations into training on resource-constrained MCUs*.

### 3.3 Importance of Dynamic Range

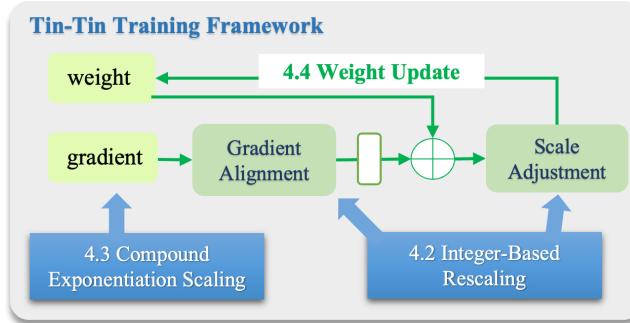
int8 has a limited representation range and precision. In a simple symmetric and uniform quantization scheme with a scaling factor  $s$ , the discrete values  $N = -128, -127, \dots, 127$  are mapped to  $sN$ . Values between  $sN$  and  $s(N + 1)$  are rounded to either  $sN$  or  $s(N + 1)$ . A smaller  $s$  reduces the rounding error but also limits the representable range, causing values outside of  $[-128 \times s, 127 \times s]$  to overflow and lead to inaccuracies.

Determining a proper scaling factor and range throughout training is crucial to balance reducing quantization error and avoiding overflow, ensuring training accuracy and stability. However, during model training, value distributions of variables (e.g., weights, activations) change dynamically. As shown in Figure 3, these ranges can differ significantly across layers depending on the layer structure (e.g., kernel size, input/output dimension), activation functions (e.g., ReLU, sigmoid) and across variables (e.g., weights vs. activations). They are also interdependent; for example, the layer output range is influenced by the distributions of both the layer input and weights, which evolve during training.

Dynamic scale adjustment is necessary to handle these variations, ensuring each variable's range is appropriately scaled to minimize overflow and maintain low quantization error. NITI [63] uses a block exponentiation scaling scheme to adjust scales dynamically but keeps weight scaling factors (determined by the weight initialization) fixed, leading



**Figure 3:** The value distributions of weights and layer outputs change significantly across layers and epochs during the training of a four-layer model for MNIST, demonstrating the need for dynamic representation ranges.

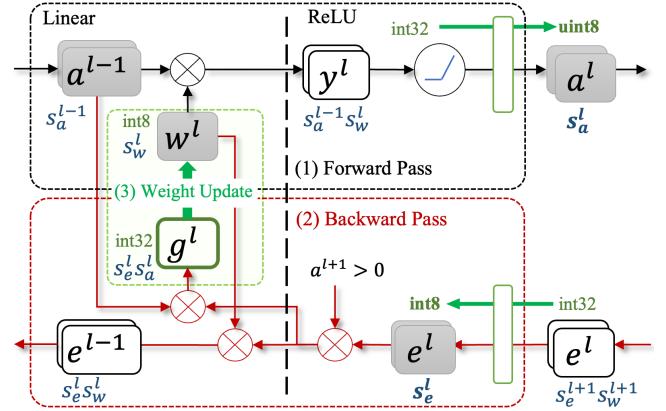


**Figure 4: Tin-Tin Training Framework**

to overflow as weights exceed the initial range after a few updates (as suggested in Figure 3). In contrast, we propose a new framework that supports dynamic weight range adjustments, reducing errors caused by improper scaling to improve overall performance.

## 4 TIN-TIN

We propose Tin-Tin, a framework designed for efficient integer-based neural network training on tiny devices like MCUs. As shown in Figure 4, Tin-Tin enhances the training process by supporting dynamic scale adjustment of integer networks through a light-weight integer-based rescaling method (Sec. 4.2) combined with a compound exponentiation scaling scheme (Sec. 4.3). An overview of the training framework and implementation specifics is provided in Sec. 4.1. We further discuss two rescaling-enabled techniques to improve the weight updates (Sec. 4.4).



**Figure 5:** The forward pass (black) and backward pass (red) of a linear layer with ReLU activation are shown. 8-bit integer operands (shaded boxes) are used for all matrix multiplications.

### 4.1 Overview

We use a linear layer with ReLU activation as an example to illustrate our approach. Figure 5 shows the forward and backward passes of a layer  $l$ , with the layer index indicated in superscript. To replace the majority of FLOPs with integer arithmetic, all matrix multiplications utilize 8-bit integer operands. These operands include the weights  $w$ , activations  $a$  from the previous layer, and backpropagated errors (i.e., gradients of the activations)  $e$ . Training data  $x$  that are not initially 8-bit integers are first quantized into 8-bit integers, then used as  $a^0$ , the input to the first layer.

*Forward Pass.* At layer  $l$ , 8-bit integer activations  $a^{l-1}$  from the previous layer (or quantized raw data for the first layer)

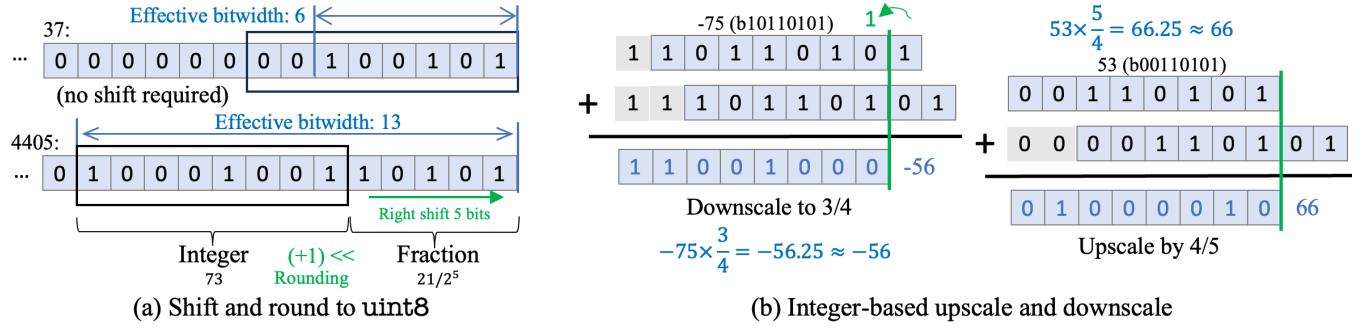


Figure 6: (a) Examples of shift-and-round to uint8. (b) Tin-Tin’s finer-grained upscaling and downscaling, achieved through shift-and-round and int8 addition.

with a scaling factor  $s_a^{l-1}$  are multiplied by the layer weights  $w^l$  in int8 format with a scaling factor  $s_w^l$ . The results of the matrix multiplication,  $y_{32}^l$ , are accumulated in int32 precision with a combined scaling factor  $s_a^{l-1}s_w^l$ . After applying the ReLU activation, the int32 values are rounded back to uint8 before being passed to the next layer (details in Sec. 4.2). We use uint8 to capture the non-negative outputs from ReLU, enhancing the precision by 1 bit than int8.

*Backward Pass.* The backward pass propagates the error back from the final loss through the integer network. At layer  $l$ , the error  $e_{32}^l$  propagated back from the next layer is initially in higher precision and is first rounded to int8 (Sec. 4.2). The error is then multiplied by the derivative of ReLU, which zeros out error terms where the original activations are negative. This adjusted error is multiplied by the layer input  $a^{l-1}$  to compute the gradient  $g_{32}^l$ , and separately multiplied by the weight  $w^l$  to derive the error  $e_{32}^{l-1}$  to send to the preceding layer. Both results are accumulated in int32 precision, with scaling factors  $s_e^l s_a^l$  and  $s_e^l s_w^l$ , respectively.

*Weight Update.* Gradient descent optimizes the model parameters by following the gradient path. Traditional gradient descent updates the weight parameters by  $w^l \leftarrow w^l - \eta/b \times g^l$ , where  $\eta$  is the learning rate and  $b$  is the batch size. However, this scaled update step is in floating-point and much smaller than the weight scale. To apply int32 gradients with a scaling factor of  $s_e^l s_a^l$  to the int8 weight  $w^l$  with a scaling factor  $s_w^l$ , we perform integer weight updates (Sec. 4.4)

$$w^l \leftarrow w^l - INT(g^l).$$

Weight updates can change the value distribution of the weight, potentially causing overflow and unwanted clipping when the resulting weight values exceed 127 (INT8\_MAX). One of our contributions (Table 1) is to allow for dynamic weight scales through integer-based scaling (Sec. 4.2) to improve the accuracy throughout the training.

## 4.2 Integer-based Upscale and Downscale

For integer-based training, it is sometimes necessary to scale the weights and intermediate results to avoid overflow when values exceed the initial quantization range. Directly multiplying integer values with a non-integer scale results in floating points and introduces FLOPs.

Simple multiplication or division by powers of two can be achieved through shift-and-round operations. This method can be used for bitwidth reduction, such as rounding int32 to int8 or uint8. Figure 6(a) illustrates this process with two examples. Given a vector of int32 values  $V$  (e.g.,  $y_{32}^l$ ), the effective bitwidth  $b^*$  is first determined, denoting the minimum number of bits required to represent the maximum absolute value  $|V|$ , calculated as  $b^* = \lceil \log_2(|V|) \rceil$ . If  $b^*$  is smaller than the target bitwidth  $b$  (8 for uint8 and 7 for int8), the int32 values can be safely truncated to 8-bit (Figure 6(a) top). Otherwise, the int32 values are right-shifted by  $b^* - b$  to retain the most significant part of the values, with the shifted-out portion treated as the fraction optionally rounded to 1 (Figure 6(a) bottom).

This right shifting effectively divides the original values by  $2^{b^* - b}$ . Similarly, integers can be multiplied by powers of two through left-shifting, which does not introduce any rounding error. However, shift-and-round can only handle rescaling to powers of two and is inefficient for finer adjustments (e.g., scaling up by 1.2 or scaling down by 0.8). To address this, we propose an efficient integer-based rescaling scheme that utilizes shift-and-round operations and int8 addition for scaling to non-power-of-two factors.

We drew inspiration from the fact that any positive values can be approximated as the sums of distinct powers of 2 and scaling by powers of 2 can be efficiently achieved through shifting and rounding. For example, to multiply the quantized integer representation by  $5/6 \approx 2^{-1} + 2^{-2} + 2^{-4}$ , instead of converting to floating points, we approximate the result by summing the integer values shifted by 1, 2, and 4 bits (i.e.,

$Q \gg 1 + Q \gg 2 + Q \gg 4$ ). Algorithm 1 outlines a binary decomposition process to determine the number of bits to shift given a scale factor  $r$ . This method enables fine-grained scaling using only shift-and-round operations and int8 addition, and can be directly applied to integers without FLOPs. This integer-based scaling method can be particularly useful in systems where floating-point operations are too expensive or infeasible, allowing for finer-grained adjustments to weight and activation magnitudes without the computational overhead.

---

**Algorithm 1** Integer-based Scaling

---

**Input:** Integer  $V$ , scale  $r > 0$

- 1:  $C \leftarrow []$ ,  $e \leftarrow \lfloor \log_2 r \rfloor$
- 2:  $c \leftarrow 2^e$
- 3: **while**  $\text{len}(C) < n$  and  $r > 0$  **do** ▶ limit # of decomposition
- 4:   **if**  $r \geq c$  **then**
- 5:      $C.\text{append}(e)$
- 6:      $r \leftarrow r - c$
- 7:   **end if**
- 8:    $e, c \leftarrow e - 1, c/2$
- 9: **end while**
- #  $e > 0$ : left shift,  $e < 0$ : right shift
- 10:  $V' \leftarrow \sum_{e \in C} \text{shift\_and\_round}(V, -e)$

---

### 4.3 Compound Exponentiation Scaling

In addition to upscaling and downscaling, which allows us to adjust scaling factors with integer operations, tracking scaling factors between quantized values and their corresponding real values throughout training is crucial. For example, when multiplying two values, the product carries a scaling factor that is the product of the two operands' scaling factors (i.e.,  $s_1 Q_1 \times s_2 Q_2 = (s_1 s_2)(Q_1 Q_2)$ ). Additionally, complex activation functions output differently based on the scaling factors of their inputs.

Scaling operations, such as reducing from int32 to int8, are intended to adjust bitwidth or the representable quantized range without altering the real values they represent. To maintain accuracy, the scaling factor must be updated accordingly. *When rescaling the quantized integer representation  $Q$  by a factor  $r$ , the scaling factor  $s$  must be adjusted by  $1/r$  to keep the real value  $sQ$  constant.* Shifting operations can be tracked using powers of two, but it is not enough for our dynamic scaling scheme (Sec. 4.2) that allows scaling to an arbitrary factor.

To address this, we introduce a compound exponentiation scaling scheme to track the scales of activations  $a^l$ , weights  $w^l$ , gradients  $g^l$ , and errors  $e^l$  for each layer  $l$  of the model during training. We use fixed upscaling and downscaling operations, as illustrated in Figure 6(b). The fixed upscaling

ratio is  $u = 4/3$  with  $3/4Q = Q \gg 1 + Q \gg 2$ , and the downscaling ratio is  $r = 4/5$  with  $5/4Q = Q \gg Q \gg 2$ .

In Tin-Tin, any scaling operations intended to adjust the scaling factors are achieved through a combination of these fixed operations and shift-and-round. Our scaling scheme employs three int8 scaling exponents: a shift exponent  $S$  (with a weight of  $2^S$  for shifting operations), and a pair of upscale and downscale exponents,  $U$  and  $D$ , for upscaling and downscaling. The combined scaling factor using these exponents is given by:

$$s = 2^S u^U r^D.$$

Each 1-bit right-shifting, upscaling, and downscaling operation increments the respective exponent,  $S$ ,  $U$ , and  $D$ , by 1. Although limiting the operations to fixed upscaling and downscaling introduces some constraints, this compound scaling scheme achieves granular rescaling, providing 12 possible scales between 0 and 1 and 8 scales between 1 and 2 within two operations of upscaling and downscaling combined with 1-bit shifting.

The memory requirement is only 3 bytes per variable. Additionally, the exponents for the product of a multiplication are simply the sum of the corresponding exponents of the operands:  $s_1 s_2 = 2^{S_1+S_2} u^{U_1+U_2} r^{D_1+D_2}$ . Algorithm 2 shows how the scaling exponents (i.e.,  $S$ ,  $U$  and  $D$ ) are calculated (line 3) and propagated from one layer to the next (line 7).

---

**Algorithm 2** Linear Layer  $l$  with ReLU - Forward

---

**Input:**  $a^{l-1}, S_a, U_a, D_a$

- 1:  $w^l, S_w, U_w, D_w$
- 2:  $y_{32}^l \leftarrow \text{ReLU}(\text{int\_mat\_mul}(a^{l-1}, w^l))$
- 3:  $S, U, D \leftarrow S_a + S_w, U_a + U_w, D_a + D_w$
- # Shift and Round**
- 4:  $sr \leftarrow \max(\text{effective\_bitwidth}(y^l) - 8, 0)$
- 5:  $a^l \leftarrow \text{right\_shift\_and\_round}(y_{32}^l, sr)$
- 6:  $S \leftarrow S + sr$
- 7: **return**  $(a^l, S, U, D)$  to the next layer

---

### 4.4 Efficient and Adaptive Update

Updating an integer network is challenging because the smallest update step is 1. This section discusses how our lightweight integer scaling techniques are applied to weight updates through gradient alignment and scale adjustment to enhance training performance.

*Gradient Alignment.* During backpropagation, a 32-bit gradient  $g_{32}$  with a scaling factor  $s_g$  is computed, and the goal is to update the int8 weight  $w$  with a scaling factor  $s_w$ . The challenge is twofold: aligning the gradient and weight scales, and determining a proper update step size.

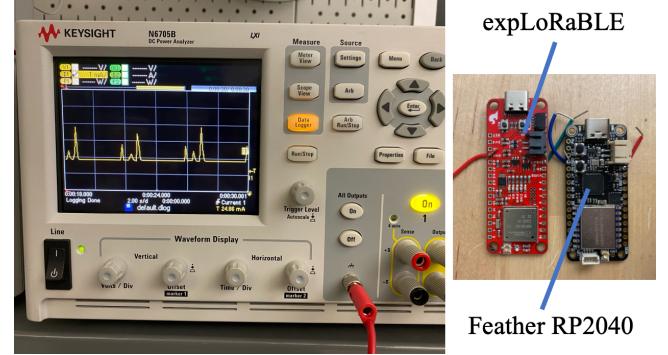
In Tin-Tin, with our compound scaling exponents, aligning the scales is achieved via a sequence of shifting, upscaling, and downscaling operations. While left-shifting can be canceled out by right-shifting the same number of bits, upscaling and downscaling are not reversible because  $u^U r^D \neq 1$  when  $U + D > 0$ . Therefore, instead of aligning the gradient scale to the weight scale or vice versa, we align both the weight and gradient to a common scale  $s$  that is reachable for both. This process is illustrated in Algorithm 3, lines 1-6. We select the smaller shifting exponent between the two because left-shifting, which reduces the shifting exponent, introduces no rounding error compared to right-shifting. The tailing zeros after left-shifting can also reduce rounding errors of the following upscaling and downscaling.

To determine the proper update step size, we draw on the rationale from LARS [70], which suggests that setting a layer-wise learning rate that maintains a certain ratio of the weights and gradients across layers improves training performance. Therefore, we determine the gradient updates at the scale of the weight by setting a fixed bitwidth difference, an update factor  $m$ , between the weight and the gradient. After aligning the weight and gradient scales, we find the effective bitwidth  $b_w^*$  of the weight and then shift and round the gradient to have an effective bitwidth of  $b_w^* - m$ , where  $m < 7$  (Algorithm 3, lines 9-12). This ensures the absolute value of the gradient updates is within  $2^{-m}$  of the weight range, then we subtract the shifted value from the weight (line 13). This method has proven effective in [63].

*Scale Adjustment.* Our proposed gradient alignment method preserves most of the gradient information by aligning both the gradient and the weight to a higher precision scale, avoiding direct truncation to low-bitwidth as done in [63]. This approach prevents *unwanted* weight overflow during updates. Since the weight is now in high precision, it needs to be rounded back to 8-bit precision. We achieve this using our integer-based rescaling method described in Section 4.2.

Given the current weight scale  $s$  and a target scale  $t$ , the goal is to find a sequence of upscaling, downscaling, and shifting operations that adjust the scale to  $t$ . This can be efficiently achieved by various combinations of our scaling operations. To facilitate this process, we pre-compute a list of scales between 0.5 and 2 that can be reached within two upscaling or downscaling operations (e.g.,  $0.64=r^2$ , achievable through two downscaling operations). We then scale down the weights by combining one of these predetermined rescale ratios with shifting operations, effectively rounding the weight back to 8-bit precision.

Our rescaling scheme offers a lightweight solution for rescaling in an integer network. However, improper use of rescaling can cause the value range to expand excessively,



**Figure 7: DC power analyzer used for power measurement and MCUs for evaluation.**

leading to poor learning performance. In Tin-Tin, we incorporate clipping techniques and maintain proper scaling factors among values in the network. This helps keep values within a reasonable range and maintain appropriate norms from layer to layer, which generally leads to good convergence. [21].

---

### Algorithm 3 Weight Update

---

```

Input:  $w^l, S_w, U_w, D_w; g_{32}^l, S_g, U_g, D_g$ 
procedure SCALE ALIGNMENT( $Q_1, S_1, U_1, D_1, Q_2, S_2, U_2, D_2$ )
1:    $S, U, D \leftarrow \min(S_1, S_2), \max(U_1, U_2), \max(D_1, D_2)$ 
2:   left_shift( $Q_1, S_1 - S$ ) or left_shift( $Q_2, S_2 - S$ )
3:   upscale( $Q_1, U - U_1$ ) or upscale( $Q_2, U - U_2$ )
4:   downscale( $Q_1, D - D_1$ ) or downscale( $Q_2, D - D_2$ )
5:   end procedure
7:    $w', g' \leftarrow \text{scale\_alignment}(w^l, S_w, U_w, D_w, g_{32}^l, S_g, U_g, D_g)$ 
8:    $b_g^* \leftarrow \text{effective\_bitwidth}(g')$ 
9:    $b_w^* \leftarrow \text{effective\_bitwidth}(w')$ 
10:   $b \leftarrow b_w^* - m$ 
11:   $g' \leftarrow \text{shift\_and\_round}(g', b_g^* - b)$ 
12:   $w' \leftarrow w' - g'$                                 ▷ Apply updates
13:   $w' \leftarrow \text{rescale}(w')$ 
14:   $b_w \leftarrow \text{effective\_bitwidth}(w')$ 
15:   $w^l \leftarrow \text{shift\_and\_round}(w', b_w - 7)$ 

```

---

## 5 EVALUATION

We evaluate Tin-Tin through two case studies of on-device training (Section 5.1), as well as simulations on the standard MNIST image classification dataset (Section 5.2).

### 5.1 On-Device Case Studies

We conduct case studies and evaluate our framework on MCUs with a focus on energy and memory efficiency. The evaluation uses two devices: (1) Adafruit Feather RP2040, featuring a Raspberry Pi Pico with a Cortex M0+ dual core

and 264 KB RAM, and (2) SparkFun LoRa Things Plus (expLoRaBLE), which uses the Ambiq Apollo3 ARM Cortex M4 with FPU and 384 KB RAM. We consider the following sensor-based learning applications:

**Motor Bearing Fault Detection.** We consider a scenario for automatic motor bearing fault detection using vibration sensors. The sensor trains an autoencoder locally on its data. To evaluate the training process, we use an extract from the CWRU bearing dataset [51], containing electric motor bearing vibration data sampled at 12 kHz in both normal and faulty states. Specifically, the fan end (FE) accelerometer data at 1797 rpm is used. A window of 32 is used as inputs. A 4-layer dense network with 24 hidden nodes and ReLU activation serves as an autoencoder. The model is trained on normal baseline data using MSE (mean-squared error) loss to learn normal conditions. During testing, the MSE (reconstruction loss) is measured on both normal and faulty conditions to evaluate the model's performance.

**Spectrum Sensing.** We implement a spectrum sensing scenario for radio signal classification. Following [59], we use an autoencoder with five dense layers for on-device radio signal classification. The dataset [59] includes I/Q samples of LTE and WiFi signals. The autoencoder is trained on LTE signals to distinguish between a mix of WiFi and LTE signals. The network architecture includes 3 hidden layers of 32, 16 and 32 nodes, using ReLU activations and MSE loss to learn LTE signals. The input comprises a history window of 16, each including real and imaginary components, phase, and amplitude values from I/Q samples, flattened into an array of 64. The model's performance is evaluated based on its ability to distinguish between LTE and WiFi signals.

**Setup.** The MCUs are programmed through Arduino. For both applications, we implement the training of autoencoders on the MCUs and compare the training efficiency using full precision floating points versus our proposed Tin-Tin method. We measure latency, memory usage, energy consumption, and record training and validation loss.

Latency for forward and backward passes is measured separately using on-board clocks and averaged over 100 training iterations. Energy measurements are conducted using a Keysight DC Analyzer N6705B (Figure 7). Since this equipment only supports sampling at 10 Hz, which is insufficient to capture all transient current behaviors, we divide the computation into forward pass, backward pass, and loss calculation. For each part, we measure the average current consumption by running the computations in a loop for 30 seconds. The measured average current is then multiplied by the measured time for each part of the computation, and the total energy costs are summed up. Each training experiment is repeated five times to obtain the average training loss.

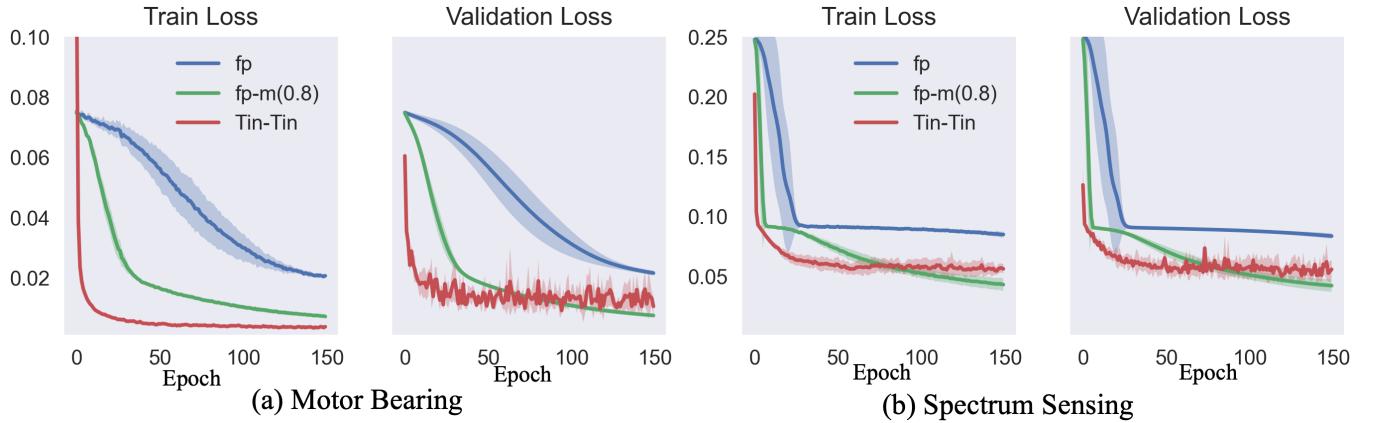
**5.1.1 Performance.** We evaluate the training efficiency and model performance after training on MCUs. For the traditional floating point implementation, a fixed learning rate of 0.1 with and without 0.8 momentum is used. We use fp-m to denote the floating point implementation with momentum. The training and validation losses, averaged across five experiments with a batch size of 128, are shown in Figure 8. The fixed learning rate without momentum converges the slowest in both scenarios and is prone to getting stuck in local optima. The use of momentum helps mitigate this issue. Tin-Tin rapidly reduces the training loss at the beginning of the training, efficiently decreasing the number of training rounds required to achieve a desired model performance.

The reconstruction loss (MSE) on unseen test data, including both normal and abnormal data (faulty motor bearings and non-LTE signals in spectrum sensing), is summarized in Table 3. A good autoencoder typically yields low loss on normal data and high loss on anomaly data. The test results indicate that fixed learning rate floating point methods are inefficient in achieving convergence in both cases. Although momentum helps to some extent, it fails to adequately detect faulty states in the motor bearing scenario, as shown by similar reconstruction errors on normal and abnormal data. In contrast, Tin-Tin enables the model to converge much faster, maintaining acceptable performance within a manageable error bound in both cases.

Additionally, we experimented with adding a check for zero values during matrix multiplication to skip MAC operations when one of the operands is zero. The average percentages of skipped MACs in the forward and backward passes are summarized in Table 3, suggesting that leveraging sparsity can significantly reduce computation costs.

**5.1.2 Resource Usage.** We compared the resource efficiency of Tin-Tin and traditional floating point implementations in terms of memory and energy consumption on RP2040 and expLoRaBLE for different batch sizes, with the results summarized in Table 4. Generally, the latency of the forward and backward passes is proportional to the batch size, as the batch of data can only be handled sequentially on MCUs. The energy consumption is almost linear with the total computation time. On devices like RP2040 without FPUs, Tin-Tin significantly reduces the training time required for each iteration by replacing FLOPs with integer arithmetic, thereby reducing computation time and saving energy by up to 83%. Tin-Tin also effectively reduces memory requirements for on-board training by using 8-bit integers instead of 32-bit floating points, reducing memory load by up to 40%.

However, Tin-Tin runs slower than floating points on the FPU-enabled expLoRaBLE, where FLOPs can be efficiently handled by FPUs within a few clock cycles. Further investigation revealed that the compiled assembly code for



**Figure 8: Training and validation MSE loss for two case study applications: (a) motor bearing fault detection and (b) spectrum signal classification. Tin-Tin rapidly reduces training loss at the beginning compared to the floating point methods.**

**Table 3: Model performance and computational saving.** Tin-Tin (int) requires significantly fewer MACs than floating point implementations, but achieves comparable or better reconstruction MSEs (lower is better on normal data, and higher is better on anomalous data).

		Reconstruction MSE		MAC Reduced	
		Normal	Anomaly	Fwd	Bwd
Motor	fp	0.0192	0.0070	-	-
	fp-m	0.0073	0.0065	-	-
	int	<b>0.0050</b>	<b>0.0107</b>	79%	30.7%
Spectrum	fp	0.086	0.124	-	-
	fp-m	0.038	0.082	-	-
	int	0.050	0.095	86%	58%

integer arithmetic is not optimized on expLoRaBLE. This demonstrates that integer-based training does not guarantee more efficiency than floating points under all circumstances, though Tin-Tin still reduces the memory requirements significantly compared to floating point deployments. Future development could benefit from system-algorithm co-design while aiming for generalizability across MCU platforms.

## 5.2 MNIST Dataset

We further evaluate Tin-Tin’s rescaling on the standard benchmark MNIST dataset using the LeNet-5 model.

**5.2.1 Rescaling Effect.** We compare Tin-Tin with NITI and floating-point updates, using a learning rate of 0.01 and a momentum of 0.8. Both Tin-Tin and NITI maintain the most significant few bits of the gradients for weight updates. In this experiment, we set  $m$ , the difference between the most

significant bit of weights and gradients (Sec. 4.4), to 4 for both Tin-Tin and NITI for a fair comparison. Figure 9 (a) shows the training loss and test accuracy averaged over five experiments when training LeNet-5 on MNIST. Both NITI and Tin-Tin, using integer updates, converge much faster than using a standard learning rate, with Tin-Tin slightly outperforming NITI. This improvement is attributed to the initial weight upscaling in Tin-Tin, resulting in larger-scale updates early in the training process, which accelerates learning compared to the weight-fixed scale in NITI.

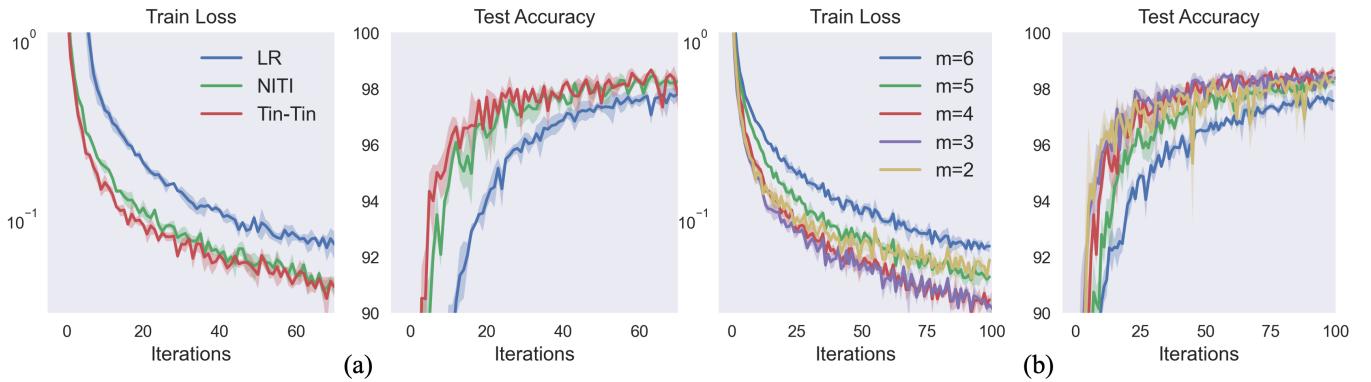
**5.2.2 Update Factor.** The update factor  $m$  is an importance hyperparameter in Tin-Tin, as it determines the size of the weight updates. A larger  $m$  increases the scale differences between the weights and updates, resulting in smaller update steps. We train on the MNIST dataset with different update factor  $m$ , and the results are shown in Figure 9 (b) and final test accuracy in Table 5. Similar to adjusting the learning rate,  $m$  controls the updating speed. A smaller  $m$  increases the update sizes, speeding up the learning process. However, when  $m$  is too small, such as 2, the maximum update value can be at most one-fourth of the weight value, leading to instability and degradation in model performance.

## 6 CONCLUSION AND DISCUSSION

In this paper, we proposed Tin-Tin, an innovative integer-based on-device training framework designed for low-power MCUs. Our approach addresses the significant challenges posed by the limited memory and computational capabilities of these devices, particularly under the lack of dedicated FPUs. By leveraging novel integer rescaling techniques, Tin-Tin efficiently manages dynamic ranges and facilitates precise weight updates using reduced-precision data types.

**Table 4: Measured average latency, energy consumption, and memory usage per training iteration for different batch sizes (BS) on RP2040 and expLoRaBLE. Tin-Tin (int) significantly reduces memory requirements, and on the Feather RP2040 significantly reduces energy requirements, compared to floating point implementations (fp) for both case studies.**

		Feather RP2040				expLoRaBLE w/ FPU			
BS		Forward Time (ms)	Backward Time (ms)	Energy (mJ)	Memory (KB)	Forward Time (ms)	Backward Time (ms)	Energy (mJ)	Memory (KB)
Motor Fault Detection	16	fp	56.46	114.02	18.51	30.27	13.67	27.74	0.60
	16	int	11.26	20.84	↓81% <b>3.43</b>	↓24% <b>22.93</b>	15.75	31.86	0.69
	32	fp	112.84	224.12	36.59	40.00	27.55	54.25	1.18
	32	int	22.42	39.98	↓82% <b>6.67</b>	↓27% <b>29.01</b>	31.01	60.44	1.32
	64	fp	225.63	443.84	72.71	59.46	55.33	114.47	2.45
	64	int	44.97	77.08	↓82% <b>13.02</b>	↓31% <b>41.04</b>	62.06	117.60	2.59
	128	fp	451.27	883.49	144.96	98.37	112.77	244.34	5.15
	128	int	89.53	151.97	↓82% <b>25.80</b>	↓34% <b>65.10</b>	131.75	254.53	5.54
Spectrum Sensing	16	fp	115.48	232.02	37.13	47.86	27.86	56.86	1.22
	16	int	21.92	44.90	↓80% <b>7.26</b>	↓31% <b>33.04</b>	30.40	68.33	1.42
	32	fp	230.89	456.56	73.45	64.58	55.69	118.65	2.51
	32	int	43.64	85.16	↓81% <b>13.99</b>	↓34% <b>42.32</b>	60.22	127.78	2.71
	64	fp	461.70	905.63	146.09	97.86	111.71	255.82	5.30
	64	int	86.72	164.02	↓81% <b>27.23</b>	↓38% <b>60.88</b>	126.64	272.45	5.75
	128	fp	923.33	1803.77	291.38	164.42	237.75	513.88	10.84
	128	int	174.06	322.10	↓82% <b>53.89</b>	↓40% <b>98.00</b>	252.45	540.79	11.44



**Figure 9: Evaluation on MNIST dataset. (a) Training loss and test accuracy using LeNet-5, comparing Tin-Tin with NITI and traditional learning rate (LR) updates. (b) The impact of update factors  $m$  on Tin-Tin’s integer updates.**

Through comprehensive real-world experiments, we validated the effectiveness of Tin-Tin across various sensor-based learning applications. Our case studies on motor bearing fault detection and spectrum sensing demonstrated the practical benefits of Tin-Tin in real-world scenarios. Specifically, the experiments showcased Tin-Tin’s ability to significantly reduce training time, memory usage, and energy consumption compared to traditional full-precision floating-point implementations.

These results underscore the potential of Tin-Tin to support energy-efficient and sustainable ML applications on edge platforms. Our framework not only extends the battery life of sensor devices but also enhances their capability to adapt to changing environments through continuous learning. Future work will include further optimizing the framework and exploring its applicability to a broader range of edge devices and learning tasks.

**Table 5: Test accuracy on MNIST after 250 iterations, with each iteration training on 15 batches of 256 data points.**

NITI		Tin-Tin			
m=4	m=2	m=3	m=4	m=5	m=6
99.2%	96.84%	97.01%	99.2%	98.98%	98.69%

## REFERENCES

- [1] Md Eshrat E Alahi, Arsanchai Sukkuea, Fahmida Wazed Tina, Anindya Nag, Wattanapong Kurdthongmee, Korakot Suwannarat, and Subhas Chandra Mukhopadhyay. 2023. Integration of IoT-enabled technologies and artificial intelligence (AI) for smart city scenario: recent advancements and future trends. *Sensors* 23, 11 (2023), 5206.
- [2] Jose M Alvarez and Mathieu Salzmann. 2016. Learning the Number of Neurons in Deep Networks. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2016/file/6e7d2da6d3953058db75714ac400b584-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2016/file/6e7d2da6d3953058db75714ac400b584-Paper.pdf)
- [3] Colby R. Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, David Patterson, Danilo Pau, Jae sun Seo, Jeff Sieracki, Urmish Thakker, Marian Verhelst, and Poonam Yadav. 2021. Benchmarking TinyML Systems: Challenges and Direction. arXiv:2003.04821 [cs.PF]
- [4] Ron Banner, Yury Nahshan, and Daniel Soudry. 2019. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems* 32 (2019).
- [5] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. 2019. Qsparse-local-SGD: Distributed SGD with Quantization, Sparsification and Local Computations. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2019/file/d202ed5bcfa858c15a9f383c3e386ab2-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2019/file/d202ed5bcfa858c15a9f383c3e386ab2-Paper.pdf)
- [6] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. signSGD: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*. PMLR, 560–569.
- [7] Yash Bhalgat, Jinwon Lee, Markus Nagel, Tijmen Blankevoort, and Nojun Kwak. 2020. Lsq+: Improving low-bit quantization through learnable offsets and better initialization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 696–697.
- [8] Osama M Bushnaq, Anas Chaaban, and Tareq Y Al-Naffouri. 2021. The role of UAV-IoT networks in future wildfire detection. *IEEE Internet of Things Journal* 8, 23 (2021), 16984–16999.
- [9] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. Tinytl: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems* 33 (2020), 11285–11297.
- [10] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations*. <https://arxiv.org/pdf/1812.00332.pdf>
- [11] Bojian Chen, Changqing Shen, Dong Wang, Lin Kong, Liang Chen, and Zhongkui Zhu. 2022. A lifelong learning method for gearbox diagnosis with incremental fault types. *IEEE Transactions on Instrumentation and Measurement* 71 (2022), 1–10.
- [12] Ning Chen, Tie Qiu, Laiping Zhao, Xiaobo Zhou, and Huansheng Ning. 2021. Edge intelligent networking optimization for internet of things in smart city. *IEEE Wireless Communications* 28, 2 (2021), 26–31.
- [13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).
- [14] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. Pact: Parameterized clipping activation for quantized neural networks.
- [15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems* 28 (2015).
- [16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs.LG]*
- [17] Ao Ding, Xiaojian Yi, Yong Qin, and Biao Wang. 2024. Self-driven continual learning for class-added motor fault diagnosis based on unseen fault detector and propensity distillation. *Engineering Applications of Artificial Intelligence* 127 (2024), 107382.
- [18] Steven K Esser, Jeffrey L McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S Modha. 2019. Learned step size quantization. *arXiv preprint arXiv:1902.08153* (2019).
- [19] Igor Fedorov, Ryan P. Adams, Matthew Mattina, and Paul N. Whatmough. 2019. *SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers*. Curran Associates Inc., Red Hook, NY, USA.
- [20] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (Portland, Oregon) (*MobiSys '22*). Association for Computing Machinery, New York, NY, USA, 464–476. <https://doi.org/10.1145/3498361.3539765>
- [21] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.
- [22] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. 2018. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems* 29, 11 (2018), 5784–5789.
- [23] 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).
- [24] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *European Conference on Computer Vision (ECCV)*.
- [25] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *The IEEE International Conference on Computer Vision (ICCV)*.
- [26] Josiah Hester and Jacob Sorber. 2017. The future of sensing is batteryless, intermittent, and awesome. In *Proceedings of the 15th ACM conference on embedded network sensor systems*. 1–6.
- [27] Yassine Himeur, Abdullah Alsalemi, Faycal Bensaali, and Abbes Amira. 2022. Detection of Appliance-Level Abnormal Energy Consumption in Buildings Using Autoencoders and Micro-moments. In *Proceedings of the 5th International Conference on Big Data and Internet of Things*, Mohamed Lazzaar, Claude Duvallet, Abdellah Touhafi, and Mohammed Al Achhab (Eds.). Springer International Publishing, Cham, 179–193.
- [28] Kai Huang, Boyuan Yang, and Wei Gao. 2023. ElasticTrainer: Speeding Up On-Device Training with Runtime Elastic Tensor Selection.

- In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services* (Helsinki, Finland) (*MobiSys '23*). Association for Computing Machinery, New York, NY, USA, 56–69. <https://doi.org/10.1145/3581791.3596852>
- [29] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2018. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research* 18, 187 (2018), 1–30.
- [30] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [31] Jeya Vikranth Jeyakumar, Liangzhen Lai, Naveen Suda, and Mani Srivastava. 2019. SenseHAR: a robust virtual activity sensor for smartphones and wearables. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems* (New York, New York) (*Sensys '19*). Association for Computing Machinery, New York, NY, USA, 15–28. <https://doi.org/10.1145/3356250.3360032>
- [32] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) (*KDD '19*). Association for Computing Machinery, New York, NY, USA, 1946–1956. <https://doi.org/10.1145/3292500.3330648>
- [33] Kunal Kejriwal. 2023. TinyML: Applications, Limitations, and It's Use in IoT & Edge Devices. Unite.AI. <https://www.unite.ai/tinyml-applications-limitations-and-its-use-in-iot-edge-devices/>
- [34] Latif U Khan, Ibrar Yaqoob, Nguyen H Tran, SM Ahsan Kazmi, Tri Nguyen Dang, and Choong Seon Hong. 2020. Edge-computing-enabled smart cities: A comprehensive survey. *IEEE Internet of Things Journal* 7, 10 (2020), 10200–10232.
- [35] Aymen Rayane Khouas, Mohamed Reda Bouadjenek, Hakim Hacid, and Sunil Aryal. 2024. Training Machine Learning models at the Edge: A Survey. *arXiv preprint arXiv:2403.02619* (2024).
- [36] Young D. Kwon, Rui Li, Stylianos Venieris, Jagmohan Chauhan, Nicholas Donald Lane, and Cecilia Mascolo. 2024. TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge. In *Forty-first International Conference on Machine Learning*. <https://openreview.net/forum?id=MWZWUyffHC>
- [37] Fengfu Li, Bin Liu, Xiaoxing Wang, Bo Zhang, and Junchi Yan. 2022. Ternary Weight Networks. *arXiv:1605.04711 [cs.CV]*
- [38] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).
- [39] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.
- [40] Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. 2021.  $\mu$ nas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 70–79.
- [41] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, and Song Han. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020).
- [42] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.
- [43] How-Hang Liu, Ronald Y Chang, Yi-Ying Chen, and I-Kang Fu. 2021. Sensor-based satellite IoT for early wildfire detection. In *2021 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–6.
- [44] Miaomiao Liu, Sikai Yang, Wyssanie Chomsin, and Wan Du. 2023. Real-Time Tracking of Smartwatch Orientation and Location by Multitask Learning. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems (<conf-loc>, <city>Boston</city>, <state>Massachusetts</state>, <conf-loc>) (SenSys '22)*. Association for Computing Machinery, New York, NY, USA, 120–133. <https://doi.org/10.1145/3560905.3568548>
- [45] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. 2020. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4876–4883.
- [46] Yao Liu, Bojian Chen, Dong Wang, Lin Kong, Juanjuan Shi, and Changqing Shen. 2023. A lifelong learning method based on generative feature replay for bearing diagnosis with incremental fault types. *IEEE Transactions on Instrumentation and Measurement* 72 (2023), 1–11.
- [47] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. 2019. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE International Conference on Computer Vision*. 3296–3305.
- [48] Konrad Lorincz, Bor-rong Chen, Geoffrey Werner Challen, Atanu Roy Chowdhury, Shyamal Patel, Paolo Bonato, Matt Welsh, et al. 2009. Mercury: a wearable sensor network platform for high-fidelity motion analysis.. In *Sensys*, Vol. 9. 183–196.
- [49] Mohsin Munir, Shoaib Ahmed Siddiqui, Andreas Dengel, and Sheraz Ahmed. 2019. DeepAnt: A Deep Learning Approach for Unsupervised Anomaly Detection in Time Series. *IEEE Access* 7 (2019), 1991–2005. <https://doi.org/10.1109/ACCESS.2018.2886457>
- [50] Maria T. Nyamukuru and Kofi M. Odame. 2020. Tiny Eats: Eating Detection on a Microcontroller. In *2020 IEEE Second Workshop on Machine Learning on Edge in Sensor Systems (SenSys-ML)*. 19–23. <https://doi.org/10.1109/SenSysML50931.2020.00011>
- [51] Case School of Engineering. [n. d.]. Case Western Reserve University (CWRU) Bearing Data Center. Retrieved June 28, 2023 from <https://engineering.case.edu/bearingdatacenter/welcome>
- [52] Zizheng Pan, Peng Chen, Haoyu He, Jing Liu, Jianfei Cai, and Bohan Zhuang. 2021. Mesa: A Memory-saving Training Framework for Transformers. *arXiv preprint arXiv:2111.11124* (2021).
- [53] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668* (2018).
- [54] Christos Profentzas, Magnus Almgren, and Olaf Landsiedel. 2022. MiniLearn: On-Device Learning for Low-Power IoT Devices.. In *EWSN*. 1–11.
- [55] Zhongnan Qu, Zimu Zhou, Yongxin Tong, and Lothar Thiele. 2022. p-meta: Towards on-device deep model adaptation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 1441–1451.
- [56] Rahul Rai, Manoj Kumar Tiwari, Dmitry Ivanov, and Alexandre Dolgui. 2021. Machine learning in manufacturing and industry 4.0 applications. , 4773–4778 pages.
- [57] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.
- [58] Philip Sparks. 2017. The route to a trillion devices. *White Paper*, ARM (2017).
- [59] Siddhartha Subray, Stefan Tschimben, and Kevin Gifford. 2021. Towards Enhancing Spectrum Sensing: Signal Classification Using Autoencoders. *IEEE Access* 9 (2021), 82288–82299. <https://doi.org/10.1109/ACCESS.2021.3087113>
- [60] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware

- neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2820–2828.
- [61] Paola Vitolo, Gian Domenico Licciardo, Luigi di Benedetto, Rosalba Liguori, Alfredo Rubino, and Danilo Pau. 2021. Low-Power Anomaly Detection and Classification System based on a Partially Binarized Autoencoder for In-Sensor Computing. In *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. 1–5. <https://doi.org/10.1109/ICECS53924.2021.9665479>
- [62] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [63] Maolin Wang, Seyedramin Rasoulinezhad, Philip H. W. Leong, and Hayden K.-H. So. 2022. NITI: Training Integer Neural Networks Using Integer-Only Arithmetic. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 3249–3261. <https://doi.org/10.1109/TPDS.2022.3149787>
- [64] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (Portland, Oregon) (*MobiSys '22*). Association for Computing Machinery, New York, NY, USA, 450–463. <https://doi.org/10.1145/3498361.3538928>
- [65] Yue Wang, Ziyu Jiang, Xiaohan Chen, Pengfei Xu, Yang Zhao, Zhangyang Wang, and Yingyan Lin. 2019. E2-Train: Training State-of-the-art CNNs with Over 80% Energy Savings. (2019).
- [66] 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*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. [https://proceedings.neurips.cc/paper\\_files/paper/2016/file/41bfd20a38bb1b0bec75acf0845530a7-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2016/file/41bfd20a38bb1b0bec75acf0845530a7-Paper.pdf)
- [67] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and Inference with Integers in Deep Neural Networks. [arXiv:1802.04680 \[cs.LG\]](https://arxiv.org/abs/1802.04680)
- [68] Md Doulotuzzaman Xames, Fariha Kabir Torsha, and Ferdous Sarwar. 2023. A systematic literature review on recent trends of machine learning applications in additive manufacturing. *Journal of Intelligent Manufacturing* 34, 6 (2023), 2529–2555.
- [69] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2022. Mandhelng: mixed-precision on-device DNN training with DSP offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking* (Sydney, NSW, Australia) (*MobiCom '22*). Association for Computing Machinery, New York, NY, USA, 214–227. <https://doi.org/10.1145/3495243.3560545>
- [70] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).
- [71] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*.
- [72] Yan Zhang, Changqing Shen, Juanjuan Shi, Chuan Li, Xinhai Lin, Zhongkui Zhu, and Dong Wang. 2024. Deep adaptive sparse residual networks: A lifelong learning framework for rotating machinery fault diagnosis with domain increments. *Knowledge-Based Systems* 293 (2024), 111679.
- [73] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).
- [74] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. 2018. Adaptive quantization for deep neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
- [75] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained Ternary Quantization. *arXiv preprint arXiv:1612.01064* (2016).
- [76] Shuai Zhu, Thiem Voigt, JeongGil Ko, and Fatemeh Rahimian. 2022. On-device training: A first overview on existing systems. *arXiv preprint arXiv:2212.00824* (2022).