Efficient Neural Networks for Tiny Machine Learning: A Comprehensive Review

Minh Tri Lê, Pierre Wolinski, Julyan Arbel

SURVEY

Efficient Neural Networks for Tiny Machine Learning: A Comprehensive Review

MINH TRI LÊ, Grenoble Alpes University, Saint Martin d'Heres, Auvergne-Rhone-Alpes, France

PIERRE WOLINSKI, Grenoble Alpes University, Saint Martin d'Heres, Auvergne-Rhone-Alpes, France

JULYAN ARBEL, Grenoble Alpes University, Saint Martin d'Heres, Auvergne-Rhone-Alpes, France

Published: 28 February 2026
Accepted: 26 January 2026
Revised: 15 January 2026
Received: 17 October 2024

Efficient Neural Networks for Tiny Machine Learning: A Comprehensive Review

MINH TRI LÊ, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, France
PIERRE WOLINSKI, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, France and LAMSADE, Paris-Dauphine University, PSL University, CNRS, France
JULYAN ARBEL*, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, France

The field of Tiny Machine Learning (TinyML) has gained significant attention due to its potential to enable intelligent applications on resource-constrained devices. This review provides an in-depth analysis of the advancements in efficient neural networks and the deployment of deep learning models on ultra-low-power microcontrollers (MCUs) for TinyML applications. It introduces neural networks and discusses their architectures and resource requirements. It explores MEMS-based applications on ultra-low-power MCUs, highlighting their potential for enabling TinyML on resource-constrained devices. The review focuses on efficient neural networks for TinyML. It covers techniques such as model compression, quantization, and low-rank factorization, which optimize neural network architectures for minimal resource utilization on MCUs. The paper then delves into the deployment of deep learning models on ultra-low-power MCUs, addressing challenges such as limited computational capabilities and memory resources. Techniques such as model pruning, hardware acceleration, and algorithm-architecture co-design are discussed. Lastly, the review provides an overview of current limitations in the field, including the trade-off between model complexity and resource constraints. Overall, this review paper presents a comprehensive analysis of efficient neural networks and deployment strategies for TinyML on ultra-low-power MCUs. It identifies future research directions for unlocking the potential of TinyML applications on resource-constrained devices.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computing methodologies** → **Neural networks**; • **Hardware** → *Power and energy*.

Additional Key Words and Phrases: Deep Learning, Tiny Machine Learning, Compression, Pruning, Quantization, Efficient Neural Networks

1 Introduction

Artificial intelligence. Over the last decade, *artificial intelligence* (AI) has revolutionized our daily experiences and technological advancements, empowering machines to perform tasks that traditionally require human-like intelligence, such as recognizing objects or speech or playing advanced games like Go. *Machine learning* (ML) is the most prominent AI approach, which trains computers to learn patterns and representations from data without explicit programming. *Deep learning* (DL) is an advanced subset of machine learning inspired by the

*Corresponding author.

Authors' Contact Information: Minh Tri Lê, minh-tri.le@inria.fr, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, 38000 Grenoble, France; Pierre Wolinski, pierre.wolinski@dauphine.psl.eu, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, 38000 Grenoble, France and LAMSADE, Paris-Dauphine University, PSL University, CNRS, 75016 Paris, France; Julyan Arbel, julyan.arbel@inria.fr, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, 38000 Grenoble, France.

ACM 2157-6912/2026/2-ART
https://doi.org/10.1145/3798276

organization of the brain, using artificial *neural networks* (NNs) to model and solve complex problems in a wide variety of fields, including language processing, protein generation, or automation.

Sensors and microcontrollers. Simultaneously, there has been an increase in the adoption and development of the *Internet of Things* (IoT), bringing new devices and applications into our daily lives. *Micro-Electro-Mechanical Systems* (MEMS) and *Micro-Controller Units* (MCUs) are essential hardware components of IoT, which allows hardware devices to collect and process information (movement, voice, temperature, pressure…) directly at the source, in their local environment, excluding the need for additional resources or external communication. Local and autonomous data processing optimizes the flow of information but inherently poses power constraints. Some applications also require continuous data processing, which puts additional power constraints. MEMS and MCUs serve as the interface to sense information between the analog and the digital world. These devices are found in a wide range of applications, including mobiles, cars, wearables, environmental monitoring, and healthcare systems. Their consumer market scales to several billion in annual sales, so a slight deviation in power constraints can result in significant costs.

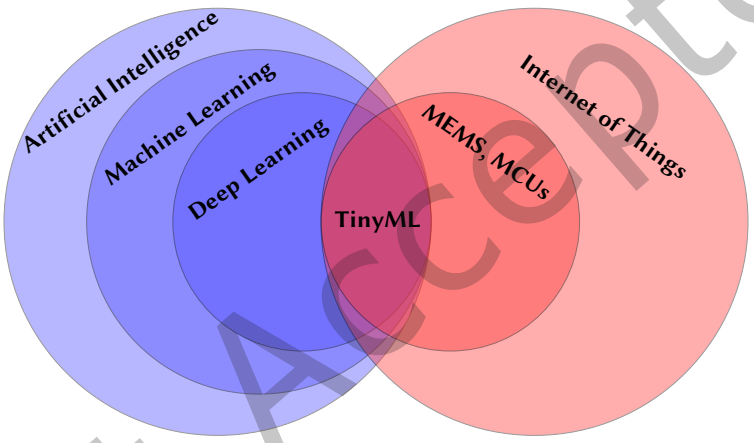


Fig. 1. TinyML as the intersection between artificial intelligence and embedded systems.

TinyML. The convergence of machine learning and IoT has sparked significant interest in both research and industry, enabling embedded hardware to process data locally and interact intelligently with their environments. This intersection has given rise to the emerging field of *TinyML*, a term first coined in 2019 by [77]; see Figure 1.

At its core, TinyML focuses on bringing the power of machine learning to resource-constrained devices such as microcontrollers (MCUs), allowing intelligent inference close to the data source with ultra-low power consumption. This paradigm unlocks applications such as gesture recognition, predictive maintenance, keyword spotting, and environmental monitoring on devices like home appliances, smartphones, wearables, and augmented reality glasses.

The key pillars of TinyML research can be summarized as follows:

- **Efficient model design**: Developing neural network architectures and algorithms that minimize memory and compute requirements while preserving accuracy. Techniques include pruning, quantization, knowledge distillation, and weight sharing.

- **Hardware-aware deployment**: Adapting models to the heterogeneous landscape of embedded hardware, taking into account the severe constraints in memory, compute power, and support for operations such as fixed-point arithmetic.
- **Toolchains and frameworks**: Providing end-to-end workflows that facilitate the translation of trained machine learning models into deployable binaries optimized for a variety of MCU platforms.
- **Applications at the edge**: Leveraging TinyML for real-world use cases that benefit from local, always-on, low-latency inference under strict power budgets.

In this context, TinyML research seeks to reduce the power consumption of neural networks at inference time, a notion we refer to as *efficiency*. Efficiency is driven primarily by minimizing memory footprint, as models must fit into highly constrained hardware, and by reducing compute operations to meet energy budgets. Typically, smaller models require less computation, making aggressive model compression and optimization essential.

Despite its promise, TinyML faces significant challenges. The exponential growth of deep learning has been driven by advances in powerful hardware, such as GPUs, which enable large-scale models with billions of parameters. In contrast, running deep learning at the edge requires fundamentally different strategies to address the tight resource constraints of MCUs. The diversity of embedded platforms further complicates deployment, demanding hardware-aware methods and frameworks.

Overall, TinyML represents a rapidly evolving interdisciplinary field that bridges embedded systems and machine learning. It presents not only technical challenges but also exciting opportunities for innovation in both algorithms and hardware, with the goal of enabling pervasive, intelligent, low-power devices.

Contributions and outline. The recent increase in research attention towards applying efficient deep learning techniques for ultra-low-power devices has led to the emergence of several review articles, which can essentially be divided into two categories.

The first category addresses *methodological aspects*:

- [74] presents an early but detailed theoretical and methodological account of quantization approaches for deep neural networks, accompanied by a brief overview of neural networks themselves. However, their scope is primarily focused on quantization as a mathematical concept and algorithm rather than the broader TinyML context, and it predates some of the most recent advances and deployment challenges specific to ultra-low-power microcontrollers (MCUs).
- [64] provides a more comprehensive and up-to-date survey of quantization methods for deep neural networks, covering topics such as quantization-aware training (QAT), post-training quantization (PTQ), and mixed-precision techniques. Nonetheless, their focus remains limited to quantization and lacks discussion on other crucial model compression techniques like pruning, distillation, or architectural considerations for TinyML deployments.
- [70] offers an in-depth analysis of knowledge distillation (KD) approaches, categorizing and comparing various KD frameworks and applications. Importantly, this survey does not address the TinyML context or constraints, and does not cover the deployment or hardware aspects required for efficient inference on MCUs.
- [87] thoroughly reviews sparsity-inducing methods, especially pruning techniques, and discusses their impact on efficiency and accuracy trade-offs. However, their analysis largely assumes deployment on general-purpose hardware (such as GPUs and mobile devices) and does not consider the extreme constraints of MCU platforms or integration with other efficiency techniques like quantization or fixed-point arithmetic.
- [4] gives a broader overview of neural network compression techniques beyond pruning and quantization, but it lacks a clear integration of neural network fundamentals, MCU hardware characteristics, and

practical deployment pipelines specific to TinyML. Their review also does not provide an entry point for readers unfamiliar with neural networks.

The second category emphasizes *applications*:

- [77] synthesizes works on TinyML applications and identifies general trends, datasets, and benchmarks. Their review serves more as a meta-analysis than a technical introduction or methodological guide.
- [179] and [167] provide rich catalogs of TinyML application domains (e.g., healthcare, industrial monitoring), existing hardware platforms, and deployment frameworks. However, these surveys primarily cater to readers interested in application landscapes and deployment ecosystems and offer little insight into neural network architectures, optimization methods, or trade-offs at the algorithmic level.

Our review begins with a general introduction to neural networks in Section 2, outlining their fundamental principles and architectures. It explores the evolution of neural networks and their applications in various domains, highlighting their computational requirements and the challenges they pose for resource-limited devices.

Then Section 3 presents a comprehensive overview of MEMS-based applications on ultra-low-power Micro-Controller Units (MCUs). It discusses the advancements in Micro-Electro-Mechanical Systems (MEMS) technology and its integration with MCUs, enabling the development of power-efficient sensing and actuation systems. The potential of MEMS-based applications in enabling TinyML on resource-constrained devices is emphasized.

The core of the review, Section 4, focuses on efficient neural networks for TinyML. This section examines various techniques and methodologies that aim to optimize neural network architectures and reduce their computational and memory requirements. It explores model compression, quantization, and low-rank factorization techniques, among others, showcasing their effectiveness in achieving high-performance inference on MCUs while maintaining minimal resource utilization.

Following the discussion on efficient neural networks, Section 5 delves into the deployment of deep learning models on ultra-low-power MCUs. It investigates the challenges associated with porting complex models onto MCUs with limited computational capabilities and memory resources. The section explores techniques such as model pruning, hardware acceleration, and co-design of algorithms and architectures, shedding light on strategies to enable efficient deployment of deep learning models for TinyML applications.

An overview of the current limitations in the field of TinyML is presented in Section 6. This section discusses the challenges faced by researchers and practitioners, including the trade-off between model complexity and resource constraints, the need for benchmark datasets and evaluation metrics specific to TinyML, and the exploration of novel hardware architectures optimized for TinyML workloads. Finally, Section 7 concludes and provides open challenges as well as insights into emerging trends and technologies that may impact the field of TinyML.

This review paper provides a comprehensive analysis of the advancements in efficient neural networks and deployment strategies for TinyML on ultra-low-power MCUs. It highlights the current state of the field and identifies future research directions necessary to unlock the full potential of TinyML applications on resource-constrained devices.

2 Neural networks

We introduce neural networks (Section 2.1), then we motivate how their theoretical properties (Section 2.2) and modern architectures (Section 2.3) are of interests in TinyML, and finally explain their implications for our work (Section 2.4).

2.1 Feedforward neural networks

The concept of artificial neural networks was introduced by McCulloch and Pitts [144] as a mathematical model to simulate the human biological neural system but was limited in its ability to learn. This laid the foundation for the perceptron model, which was the first neural model capable of learning and classifying linearly separable

data [170, 175]. In turn, the backpropagation [171] and gradient descent algorithms [11, 118] were developed to allow efficient training of multi-layer perceptron (MLP) that is able to classify non-linear inputs. The MLP is a type of feedforward neural network that consists of alternatively stacking multiple layers L of neurons and non-linear functions ϕ [91, 171] as represented in Figure 2. These layers include an input layer, one or more hidden layers, and an output layer. Stochastic gradient descent (SGD) and backpropagation algorithms, and progress in hardware computation have enabled the revolution in the field of neural networks, leading to the modern era of deep learning algorithms [117], for example capable of achieving state-of-the-art performance on ImageNet [109].

Formally, a neural network can be defined as a function f and a directed, weighted graph composed of nodes (neurons) and edges (connections between neurons) with associated weight parameters W, bias B, where inputs x are propagated forward in the graph to produce an output y. The objective of the neural network f defined as:

$$\begin{aligned} h^{(0)} &= x, \quad y = f(x) = h^{(L)}, \\ h^{(l)} &= \phi^{(l)}\left(W^{(l)} h^{(l-1)} + B^{(l)}\right) \quad \text{for } l = 1, \dots, L, \end{aligned} \tag{1}$$

is to approximate some function f^* mapping an input vector x to an output vector y by learning the weight matrices $W^{(l)}$ and the bias vectors $B^{(l)}$ [69].

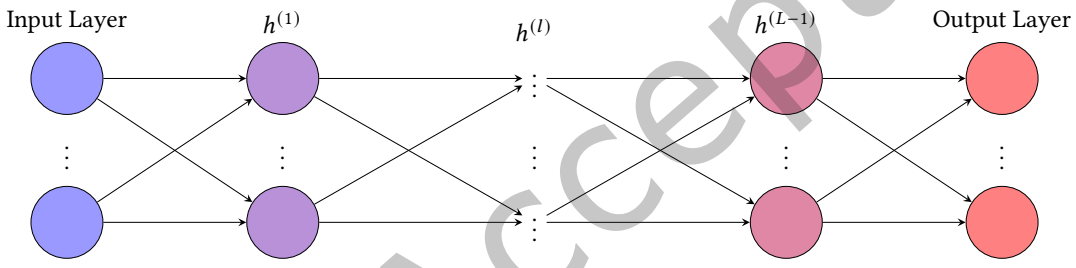


Fig. 2. Feedforward neural network.

Neural networks have interesting theoretical and practical properties, as we will see in the next sections.

2.2 Properties

Neural networks possess powerful theoretical properties that stand out from standard machine learning approaches, making them of great interest for a wide range of applications.

Expressiveness and generalization. Neural networks are universal approximators: Cybenko [42], Hornik et al. [88] have theorized that a sufficiently wide hidden layer is able to approximate any continuous function on a compact set to an arbitrary level of precision. More recent work by Lin and Jegelka [129] has extended the universal approximation theorem to residual neural networks [ResNets, 80], proving that a sufficiently deep neural network with one-neuron hidden layers with residual connections has enough expressive power to approximate any continuous function.

Another crucial property of neural networks, related to generalization in statistical learning theory [204, 207], is their ability to generalize to new data with fewer examples than parameters, even with very large models [102, 122], and they are even capable of labelling random data [225]. This overparameterization results in a highly-dimensional non-convex space and redundancy, but results also in higher quality and quantity of local minima [37]. This implies that the optimization process has a higher chance of not getting stuck in a bad local minimum compared to small-size networks.

Over the recent years, research has revealed that enlarging the model size beyond the quantity of training examples can lead to a peculiar trend in test error: it may initially peak at a certain point of model complexity, then unexpectedly begin to decline again. This intriguing behavior has been termed *double-descent* by [17], who demonstrated its presence across various machine learning models, including a two-layer neural network. Further investigation into this phenomenon has been conducted by [151], who extensively explored double-descent in deep neural network models. They found that this trend can manifest when altering the model width or the number of optimization iterations. Additionally, they observed instances of the double-descent phenomenon being influenced by dataset size: larger datasets sometimes leads to inferior test performance.

Despite these findings, the underlying reasons behind the occurrence of double-descent in machine learning models and the specific inductive biases responsible for it remain incompletely understood. Nonetheless, it is crucial to consider this phenomenon when devising strategies aimed at enhancing generalization capabilities.

Issues with expressiveness and generalization properties of TinyML. These useful properties are mainly verified for *large* neural networks. For instance, a two-layer neural network is ensured to be a universal approximator if its hidden layer is wide enough. On the generalization side, experimental and theoretical findings tend to show that larger models generalize better. However, in the TinyML setup, the neural networks are far from these large, well-studied models. Therefore, obtaining good training and generalization results in TinyML is very challenging.

After this brief overview of the theoretical properties of deep learning, we will now explore which modern deep learning architectures are commonly used in practice and why.

2.3 Modern deep learning architectures

Although in the modern deep learning era, the hardware progress can allow supporting the given high volume of computation and data, the design of the architecture is critical to the final performance and depends on the applications.

Developing and finding new neural network architectures is of great interest in research to surpass the state-of-the-art. Most of these state-of-the-art architectures are variations and combinations of the ones we present below. Table 1 provides a summary of standard architectures used in modern deep learning, and their strengths and weaknesses.

Fully-connected layers. Fully-connected (FC) layers, also known as dense layers were the first type of layers used in neural networks, specifically in MLPs as presented in Section 2.1 and depicted in Figure 2. In a fully-connected layer, each neuron is connected to all the neurons in the next layer and process each input independently by applying a non-linear transformation. They are often used toward the end of the model to aggregate the higher-level features from the previous layers and make the final predictions. The simplest form of a fully-connected layer is a weighted sum, which makes them very general and not specialized to any particular application. Thus, they are building blocks of modern deep learning architectures. However, they are prone to overfitting, and may poorly perform on spatial or temporal data.

Convolutional neural networks. Convolutional neural networks (CNNs) are commonly used as feature extractors, showing their strength in processing spatial structures, such as images [109], videos [183], or signal processing [3, 68]. As the name suggests, they consist of applying convolutional operations using filters, also called kernels on the input in 1D, 2D or 3D, and are shared across the spatial dimensions. They are often stacked all together with max-pooling, to summarize a group of values by their maximum [109], batchnorm to normalize activations and facilitate training [96], and ReLU activation function. Compared to FC layers, this design allows CNNs to efficiently learn spatial hierarchical structures and detect local to global patterns, such as edges, shapes, and textures. In addition, the weight-sharing aspect reduces the number of parameters and makes them more robust

to spatial translations and distortions. Some classic CNN architectures are AlexNet [109], VGGNet [184], or GoogleLeNet [193], each using increasing network depths, thereby large model size.

Thus, in modern deep learning architectures, CNNs are often found in the early stages of the network serving as powerful feature extractors, but they have shown limitations in learning with sequential data structure or modeling long-range dependencies [132, 182].

Recurrent neural networks. Recurrent neural networks (RNNs) are specialized layers for modeling sequential data [47, 171], such as signals [3, 72], speech [227] or text [10]. Compared to CNNs, they are able to model longer temporal contexts by keeping a description of previous contexts because each output directly depends on previous inputs. This is of particular interest for sensor-based applications that inherently deal with sequential data.

The building block of a RNN can be defined as follows [47]:

$$h_t = \phi_h(W_h[h_{t-1}, x_t] + b_h), \quad y_t = \phi_y(W_y h_t + b_y), \tag{2}$$

where x_t is the input, h_t is a shared internal state, serving as a *memory* at time t, b_h and b_y are bias terms, and ϕ_h, ϕ_b are activation functions. However, they are difficult to train because of the effects of the vanishing or exploding gradient when the sequence is long [19]. Then long-short term memory [LSTM, 62, 63, 86] and gated-recurrent units [GRU, 40] layers have been designed to alleviate the limitations of the simple RNN.

They are based on two kinds of memory updates:

- *Leak* memory updates, that are *progressive* updates of the current memory: $h_{t+1} = h_t + \phi(h_t, x_t)$,
- *Gate* memory updates, that are *context-dependent* updates of the memory: $h_{t+1} = \alpha h_t + (1 - \alpha)\phi(h_t, x_t)$,

where α can be a scalar or the output of a gated function $g(h_t, x_t) \in [0, 1]$ as in GRU or LSTM. Note that the "gated" mechanism is a specific form of the attention mechanism [205], allowing it to focus its attention on specific inputs depending on the context.

In particular, LSTM has three gates (input, forget, and output) and has two hidden temporal streams, one corresponding to the RNN stream of Equation (2), and another auxiliary stream used to compute α, thus controlling the number of updates.

GRU is a simplified version of LSTM (update and forget) as well as one hidden temporal stream h_t, which has shown performance close to LSTM with a lower power footprint [28].

However, RNNs are limited in handling spatially structured data and processing sequences in parallel. This is because RNNs process input one time step at a time, Equation (2).

Residual neural networks. Residual neural networks (ResNets) were introduced in He et al. [80]. They provide each layer with direct feedback from distant previous layers to minimize the loss of gradient information during the backpropagation in deep networks. Although ResNets have shown state-of-the-art performance in computer vision [103], they are typically on the scale of millions of parameters [147] and are more commonly applied on deep networks, which is not suitable for TinyML hardware.

Transformers and attention-based models. Transformers are attention-based models introduced in Vaswani et al. [205] that surpass state-of-the-art performance on large-scale natural language processing tasks or computer vision tasks [131]. They allow the models to focus their attention on each token of the input sequence (local) with respect to other tokens (global). This design addresses the limitations of CNNs and RNNs as stated previously because Transformers can process long-term dependencies and sequences in parallel. Although they are a great success and generate interest, they require a large amount of data, and a power footprint for both training and inference, even more than ResNets, which makes them difficult candidates for TinyML. However, Transformer and attention-based models are increasingly used in TinyML: recent works have explored various strategies to adapt Transformers for TinyML, including their application in anomaly detection for IoT and embedded systems [16, 181], optimizing deployment on low-power microcontrollers [100, 218], employing quantization and

knowledge distillation to reduce model size and improve efficiency [192], and investigating linear Transformer architectures tailored for constrained hardware [178].

Table 1. Summary of standard architectures used in modern deep learning.

Layer & Definition	Strength	Weakness
FC: Connects all neurons in-between layers	High-level aggregations	Overfitting, not specialized
CNN: Conv. operations with shared parameters	Local and global spatial patterns	Struggles with sequences
RNN: Processes sequences with a hidden state	Temporal dependencies	Struggles with spatial patterns
ResNets: Deep nets with residual connections	Eases training deep networks	Large model size, expensive
Transformers: Self-attention for input relationships	Long-term local and global patterns	Large training data and power footprint

Activation functions. Activation functions in deep learning introduce non-linearity to the model, enabling deep learning models to achieve higher levels of expressiveness and create more complex decision boundaries. This non-linearity is essential for processing real-world data, characterized by diverse and often non-linear features, effectively capturing intricate relationships within the data. Table 2 references standard activations used in modern deep learning.

Table 2. Reference table of standard activation functions, where Φ represents the standard Gaussian cumulative distribution function.

Name	Definition	Notes
ReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$	Returns identity if positive, else 0
Leaky ReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases}$	Allows small negative values
PReLU	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha_i x, & \text{otherwise} \end{cases}$	Per-neuron learnable α_i values
ELU	$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$	Exponential smoothing, $\alpha > 0$
GELU	$f(x) = x\Phi(x)$	Smooth approximation of ReLU
Mish	$f(x) = x\tanh(\ln(1 + e^x))$	Smooth and non-monotonic
Swish	$f(x) = x\sigma(x)$	Smooth, non-monotonic
Hard Swish	$f(x) = x\frac{\max(0,\min(6,x+3))}{6}$	Efficient Swish variant for TinyML
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	Returns value in range $(-1, 1)$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	Returns value in range $(0, 1)$
Hard Sigmoid	$f(x) = \max(0, \min(1, \frac{x+1}{2}))$	Fast approximation of sigmoid
Softmax	$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$	Returns class probabilities

Regularization. In Section 2.2, we have seen that neural networks possess interesting generalization properties. We will now explore popular regularization choices that help with generalization in practice.

As in standard machine learning, regularization can help neural networks to generalize better to unseen data, and make them less complex. Regularization techniques can either be of two forms, based on whether or not they directly alter the objective function:
Explicit regularization:

- L_1 penalizes the absolute values of the weights, encouraging sparsity, and thus simpler models,
- L_2 penalizes the squared values of the weights, constraining their magnitude, and thus encourages smoother and simpler models.

Implicit regularization:

- Dropout [190] as an average of probabilistic architectures where each dropout-realization results in a different sub-network [57],
- Batch normalization limits the range of values and adds noise to the activation, preventing the model from memorizing the training data too well [23, 96],
- Early-stopping prevents the model from becoming too specialized during training [22, 186],
- Data augmentation increases the size and diversity of the training set, which helps the model learn more robust features [182],
- Random noise injected into the input (also a form of data augmentation) [69],
- Noise introduced by SGD optimization [161, 162].

Most of these regularization methods add negligible computation cost and help with generalization performance. In this section, we provided a brief overview of the layers used in modern deep learning and discussed which have the most potential for low-power hardware applications.

2.4 From large deep learning models to TinyML

In this section, we give an overview of the recent trends of deep learning model sizes, then we explicit the challenges of TinyML based on the neural network theory (Section 2.2) and practices (Section 2.3), and motivate our interest to use them for TinyML.

Trend in deep learning models. Since the first AlexNet model was trained on a graphic processor unit (GPU) [109], we entered the modern era of deep learning where the limits of the state-of-the-art are regularly pushed on numerous complex tasks. Meanwhile, deep learning models are geared towards exponential increases in model size. As of 2023, the GPT4 model [157] is said to be even larger than the GPT3 model with 175 billion parameters ($\approx$ 800GB) [25], being about 2800 times larger than AlexNet size in just over 8 years. Although model performance can benefit from overparameterization, large neural networks have been shown to have high redundancy [53, 78]. Denil et al. [45] estimated that in some cases only about 5% of the total parameters are critical to the final output decision. Thus, we can see that these models fail in terms of *algorithm efficiency*, where the objective is to achieve a task with minimal effort. This raises questions on how to train more efficient models and also suggests the existence of smaller but viable models.

Trend in efficient deep learning models. A new wave of efficient deep learning models emerged, such as SqueezeNet [94], MobileNet V1, V2, and V3 [89, 90, 176], or EfficientNet [195], ranging from one to five million parameters, entering the scale of the feasibility on mobile devices. These new models can achieve up to a 510-time model size reduction compared to AlexNet [195] with equal performance. In general cases, model sizes are of the order of at least 10^6.

Trend in ultra-low-power deep learning models. Although mobile-sized models show a great shift toward efficient deep learning architectures, they are still too large for deployment on microcontrollers [14, 126, 130]. Deep learning on microcontrollers [201] is an alternative paradigm that is still at an earlier stage compared to mobile-size research, where the term TinyML has been first appearing in 2019 [77]. However, there has been a success in the deployment of neural networks on MCUs on audio classification tasks [51, 130, 227] by using efficient CNNs, RNNs, or NAS [14]. In Lin et al. [130], they succeeded in deploying a person detection model with less than 1MB memory. In general cases, model sizes must be of the order of less than 10^6 and less than 1MB. These models reach a memory size of under 512 kB or even 256 kB, entering the scale of microcontroller hardware. The severe resource limitation of MCUs present unique requirements and need the design of dedicated workflow and tools to enable end-to-end deep learning pipelines. Table 3 provides a summary of example model sizes for each platform we reviewed.

Table 3. Comparison of representative deep learning model sizes across cloud, mobile, and MCU platforms.

Platform	Model	Parameters	Model size
Cloud	Inception-v3	$> 10^7$	$>$ 100 MB
Mobile	MobileNet-v3	10^6	$>$ 1 MB
MCU	MCUNet	$< 10^6$	$<$ 1 MB

Motivations. Neural networks are powerful algorithms that can operate with an end-to-end approach in terms of algorithm design: labelled data, automated feature extraction and modeling, and deployment, for a wide range of applications. This makes them a great class of algorithm candidates for MEMS-based applications relying on signal processing. Unfortunately, the expressiveness and generalization ability of neural networks are dependent on their size, which makes them inherently complex and "black box" functions that are analytically difficult to interpret and design. However, they are mostly composed of very primitive operations, see Equation (1): *multiplications and additions*, which are accessible to any microcontrollers. Concerning the non-linear activations, some are very straightforward, such as ReLU [56, 150] or LeakyReLU [142], while other activations like tanh or sigmoid pose more challenges due to their computational complexity.

Moreover, prior literature has shown that it is *possible*, albeit *challenging*, to design and deploy small enough neural networks on resource-constrained microcontrollers. Therefore, following the trend of efficient deep learning models to reduce their inherent power footprint, we are interested in pushing the state-of-the-art of low-power footprint models to make them viable to microcontrollers, without degrading performance. Additionally, deep learning models in practice are commonly overparameterized [45], so the field of deep learning will benefit from more contributions to the design and deployment of more efficient and accessible neural networks.

To summarize, we provided background on neural network theory and practice, their limitations and challenges, and why they are a great research interest for MEMS-based applications running in ultra-low-power settings.

Next, we explore the literature on specialized methods to design efficient deep learning models for TinyML in Section 4, but we must first provide the necessary background on embedded hardware, which we will reference throughout our work in Section 3.

3 MEMS-based applications on ultra-low-power microcontrollers

We provide a brief overview of MEMS and MCU hardware technology (Section 3.1) to understand the scope of applications (Section 3.2) and their intrinsic challenges for deep learning (Section 3.3).

3.1 Overview

MEMS and MCUs. MEMS are miniaturized (microscale dimensions) sensors and actuators omnipresent in a wide range of electronic devices, as they convert physical and analog information into digital inputs about their local environment [113, 230], that can be processed by MCUs in real-time. Some examples of MEMS are accelerometers, microphones, or pressure sensors. Table 5 provides examples of different sensor types and their applications. Thus, they provide an interface to sense real-world information from hardware to software.

MCUs are miniaturized computers that are non-invasive ($\sim$ 1 mm^2 silicon area), cheap ($\sim$ 1\$), low-power ($\leq$ 0.5 W), and are dedicated to performing one task for months or even years within a device [14, 60]. MCUs are composed of connectors, input/output interface, on-chip storage (ROM), volatile memory (SRAM) for intermediate data, and a CPU with a frequency usually below the 10^3 MHz range [14]. With over 250 billion MCUs already in use, forecasts predict a volume of 38.2 billion in 2023 alone [130]. In this context, we emphasize that even a small difference in the power footprint between low-power hardware targets can translate to several billions of dollars in savings for the consumer market. This is exemplified by the 2\$ difference observed between the low-end of MCUs in Table 4. Even between MCUs, there are several orders of magnitude in terms of low power (Table 4). For example, the Cortex-M4 only consumes 0.1W, yet it still represents a target that is 1500 times more power-hungry and 20 times more memory (SRAM) capacity compared to the Cortex-M0+. Additionally, it is three times more costly for consumers. Consequently, it is important to highlight the strong industrial incentive to target the low-cost and low-power consumer market as much as possible with tiny hardware targets. By focusing on the power scale between these targets, we can realize billions in cost savings and other benefits that low-power MCUs offer for the consumer market.

Applicability. Sensing data at the edge allows for offline operations, as opposed to using online cloud computing, always-on and real-time processing, no network latency, limited energy overhead, and inherent privacy. MCUs are ubiquitous in modern electronic devices, including cars, mobiles, TVs, and cameras. Their high volume in the consumer market and wide applicability reinforce the significance of research and industry efforts in TinyML applications.

ARM processors dominate TinyML deployments thanks to their mature ecosystem, widespread availability, and extensive software support. However, RISC-V is emerging as an attractive alternative due to its open, extensible Instruction Set Architecture (ISA) that allows hardware customization for specific TinyML workloads (e.g., specialized instructions or accelerators). While ARM offers excellent performance, power efficiency, and broad toolchain support out of the box, RISC-V provides greater flexibility for application-specific optimizations but requires more effort in ecosystem development and toolchain maturity.

A number of PhD theses recently considered using RISC-V in TinyML: Gao [59] presents LiteQAIRISC, a customizable RISC-V emulation for TinyML, while Maras [143] develops fine-grained mixed-precision RISC-V ISA extensions; additionally, Scherer [177] demonstrate hardware-software co-design on RISC-V for TinyML inference. A surge of research articles were also devoted to the use of RISC-V in TinyML in the recent years: Zoni and Galimberti [232] propose a fixed-point cost-effective solution for RISC-V MCUs; Garofalo and Benini [61] leverage RISC-V extensibility for flexible TinyML SoCs; Verma et al. [206] propose Extrem-Edge, a RISC-V-based architecture for energy-efficient ML inference; Hassan and Sagahyroon [79] provide a comprehensive review of RISC-V's role in AI-IoT; Jung et al. [100] optimize Tiny Transformers on RISC-V MCUs; Z Huang [223] present RIOT-ML benchmarking on RISC-V and ARM; M Beltrán-Escobar [140] review TinyML vision systems on RISC-V vs ARM; Ottaviano et al. [158] introduce Cheshire, a lightweight customizable RISC-V platform for TinyML accelerators.

While the open Instruction Set Architecture (ISA) of RISC-V promotes design freedom, its software ecosystem remains comparatively immature relative to ARM. In practice, developers still face limitations in compiler optimizations, peripheral driver support, and integrated development environments (IDEs) for embedded AI.

Table 4. Comparison of hardware for **Cloud**, **Mobile**, and **TinyML** platforms [14, 173]. The three architectures studied in Section 6 are highlighted in blue.

Platform	Architecture	Memory	Storage	Frequency	Power	FLOPS	Price
Cloud	*GPU*	*HBM*	*SSD/Disk*				
Nvidia V100S	NVIDIA Volta	32GB	TB~ PB	1.2−1.3GHz	250W	~16.4G	14500$
Mobile	*CPU*	*DRAM*	*Flash*				
Galaxy Note 20	Kryo 585	8GB	128GB	1.8−3.1GHz	~8W	1.2T	550$
TinyML	*MCU*	*SRAM*	*eFlash/ROM*				
SAME70Q21B	Cortex-M7	384kB	2048kB	300MHz	0.3W	~432M	5$
SAMG55J19	Cortex-M4	160kB	512kB	120MHz	0.1W	~180M	3$
Newport	Cortex-M0+	8kB	16kB	6.14MHz	70μW	N/A	1$
Newport	eDMPv1	4kB	16kB	6.14MHz	66μW	N/A	1$
HiFive1 Rev B	RISC-V RV32IMAC	16kB	512kB	320MHz	0.14W	~215M	50$
VEGAboard	RISC-V RV32IMC	8kB	64kB	100MHz	0.1W	~80M	20$

Frameworks such as TensorFlow Lite Micro [44, 212] and Tensor Virtual Machine (TVM) [32] have recently begun adding native RISC-V back-ends, yet their coverage of instruction extensions (e.g., Digital Signal Processing (DSP), vector, and bit-manipulation) is partial. Consequently, efficient TinyML deployment on RISC-V currently requires low-level toolchain customization or hardware-software co-design. Nonetheless, this flexibility also provides a unique opportunity for research into specialized micro-architectural extensions tailored for inference, such as custom multiply-accumulate pipelines or quantization-aware operators.

In comparison, ARM-based MCUs benefit from a long-established toolchain and ecosystem maturity, including standardized Cortex Microcontroller Software Interface Standard for Neural Networks (CMSIS-NN, 112) libraries, optimized kernels, and broad support across mainstream frameworks. As a result, porting and benchmarking TinyML workloads are typically faster and more predictable on ARM. However, the openness of RISC-V encourages collaborative innovation among academia and industry, making it a fertile ground for exploring new accelerator topologies and instruction-set extensions. Future work could benchmark identical TinyML models across both architectures, quantifying differences in latency, energy per inference, and model size to better characterize the trade-off between ecosystem maturity and architectural flexibility.

In this review, we target the most *extreme low-end range* of MCUs, with less than 8kB of RAM and 10MHz processing speed for extreme low-power deep learning inference. Therefore, we aim to push the hardware limit that is currently not considered in the state-of-the-art for embedded deep learning. In particular, we focus on the common ARM Cortex-M series microcontrollers [221], and particularly the Cortex-M0+ and M4 (Table 4), or the eDMPv1 depending on the application.

Variety of MCU platforms. A key challenge for general TinyML solutions is the heterogeneity of embedded hardware platforms. Microcontrollers vary widely in compute capabilities, memory capacity, support for floating-point versus fixed-point arithmetic, instruction set architectures (e.g., ARM Cortex-M series vs. RISC-V cores), and availability of hardware accelerators. As a result, models and deployment pipelines often need careful tuning and customization to match the specific constraints of a given target device. This hardware diversity limits the portability of TinyML solutions and complicates the development of universally applicable frameworks and tools, underscoring the need for adaptive techniques and hardware-aware optimization methods.

3.2 Scope of applications

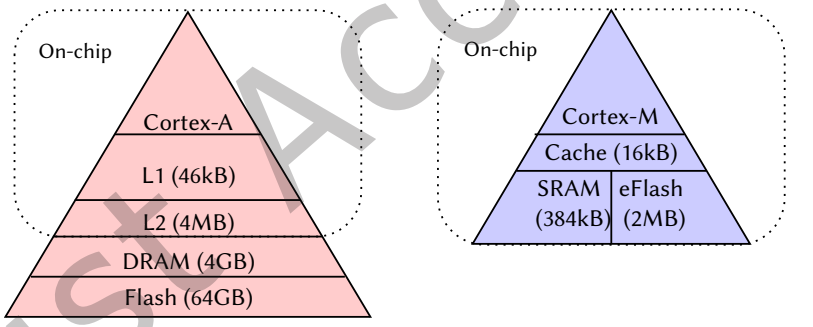
As previously stated, the ability to embed neural networks at the edge can already benefit a wide variety of applications and can potentially lead to completely new types of products [101].

Common applications are image detection, gesture recognition, such as human activity recognition (HAR), or keyword spotting. Note that these are all wireless applications, that must operate in real-time and are always-on. In this context, the device returns a decision at all times, so it is expected to provide a seamless user experience (e.g., not missing any user intention (false negatives) or over-triggering (false positives)). Their sensor types and target devices are specified in Table 5.

Table 5. Example of sensor applications and their target MCU devices.

Sensor types	**Applications**	**Target devices**
Accelerometer, Gyroscope, Magnetometer	Human activity recognition, gesture recognition, motion detection, voice detection, predictive maintenance	ARM Cortex-M0+
Pressure	Fingerprint detection	ARM Cortex-M0+, Cortex-M4
Microphone	Sound classification, keyword spotting	ARM Cortex-M4, Cortex-M7

3.3 Challenges of ultra-low-power hardware



(a) Mobile processor (b) ARM Cortex-M7 microcontroller

Fig. 3. Illustration of memory hierarchies for (a) a mobile processor and (b) an ARM Cortex-M7 microcontroller (right). The microcontrollers process all computation and data transfer on-chip.

Compared to mobile devices, the all-on-chip design, as shown in Figure 3, allows the processing of data at the closest location to the source, resulting in lower communication latency and lower power consumption. Thus, this is ideal for real-time and low-power constraints. However, it also makes them inherently constrained because additional memory cannot be extended with an SD card for example.

Moreover, Table 4 highlights that the Cortex-M0+ and M4 are among the most resource-constrained devices, with the Cortex-M0+ lacking support for floating-point operations. Consequently, we restrict to fixed-point (in contrast to floating point) values (Figure 4) and arithmetic which approximates real-values and computations

[145], to comply with the inherent hardware and energy constraints of MCUs. Floating-point to fixed-point conversion requires a scaling factor of a power of two, which can be inferred as a simple bit shift and rounding as follows:

$$Q(F, n) = \lfloor F * 2^n \rceil, \quad F(Q, n) = Q * 2^{-n}, \tag{3}$$

where Q and F are the fixed point and floating point numbers, respectively, and n is the number of bits. In practice, this means that we are limited to *integer-only operations*. Thus, only primitive operations like bit-manipulation, boolean operators, and basic additions or multiplications are supported in contrast to computationally intensive operations, such as explicit division or exponentiation. Additionally, the memory is typically the first bottleneck, so we seek lower-bit precision parameters than 32-bits, but this may increase the risk of overflow, or numerical precision loss and thus erroneous inference. From a hardware point-of-view, restricting to integer-only inference removes the need for a floating-point unit, which saves silicon area for each embedded chip, and thus billions of dollars of annual savings.

(31)	(30 29 28 27 26 25 24 23)	(22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)
sign (1-bit)	exponent (8-bits)	significand (23-bits)

(a) Single precision floating-point 32-bit representation from IEEE754 [95].

(31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16)	(15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)
integer part (m-bits)	fractional part (n-bits)

(b) Fixed-point Q16.16 (Qm.n) on 32-bit representation.

Fig. 4. Floating-point and fixed-point 32-bit representations. Floating-point (b) allows a dynamic range (minimal to maximal possible value) of roughly $[-10^{38}, 10^{38}]$, compared to fixed-point (a) $[-2^{m-1}, 2^{m-1} - 2^{-n}] \approx [-2^{15}, 2^{16}]$, which is approximately a 10^{33} smaller range [154]. The smallest resolution (step between each consecutive representable value) of floating-point is $\approx 10^{-38}$ while it is $[2^{-n}] \approx 10^{-5}$ for $n = 16$ for fixed-point.

After a comprehensive review of the literature, the Cortex-M0+ and eDMPv1 appear to be one of the most resource-constrained platforms on which successful implementation of state-of-the-art deep learning has been reported [14, 173, 227]. Zhang et al. [227] deployed a 70kB keyword spotting application on an Arm Cortex M7, while Banbury et al. [14] deployed the same application on an Arm Cortex-M4 with a higher accuracy.

Furthermore, embedded hardware has a very heterogeneous ecosystem because specifications may differ from one manufacturer to another, and even between new series of the same brand, making it challenging to find common tools and approaches that are widely supported.

Therefore, the ultra-low-power hardware context presents a unique set of challenges due to their inherent resource limitations. Addressing these challenges poses high research and industry potential value and can lead to transformative advancements in real-time and low-power applications across numerous domains.

To summarize, in Section 2 and Section 3 we provided background on neural networks, and low-power sensors and motivated the challenges and objectives of our TinyML. We will now examine the literature on methods (Section 4) and tools (Section 5) to design and deploy efficient neural networks for MEMS-based applications.

4 Efficient neural networks for TinyML

Building upon the concepts and motivations surrounding neural networks and embedded systems introduced in the previous sections, we now turn our attention to their intersection: TinyML.

This emerging field aims to combine the powerful benefits of neural networks with the cost-effectiveness of ultra-low-power devices with limited power, memory, and processing capabilities. Given the constraints of TinyML, developing efficient neural network architectures and algorithms is essential. In light of the growing efforts in this area, there is an increasing need for methods that can effectively scale to the most challenging embedded hardware, particularly in the context of MEMS-based applications.

In this section, we explore the methods available to train and design efficient neural networks for deployment on MCUs, enabling the deployment of intelligent applications on low-cost devices. In particular, we emphasize that quantization is the most critical method since it is a mandatory step in deploying models on microcontrollers.

Efficient RNNs. Sensor applications mainly process time-related data continuously, so we are naturally interested in standard RNN layers, such as RNN [47, 171], GRU [40], LSTM [86]. Arık et al. [8], Bhardwaj et al. [21], Lu et al. [139] have used convolutional recurrent neural networks (CRNNs) with a GRU or LSTM as the recurrent layer for keyword spotting or motion recognition applications for low-power and real-time inference, which matches our target applications and environment. The CRNN architecture offers strengths both in feature extraction, and time sequence processing, as well as compatible size for our target hardware [21].

In particular, Arık et al. [8] empirically showed that GRU layers offer better size-performance tradeoff over LSTM in keyword spotting applications, which is our most demanding use case.

Moreover, there have been research efforts to find efficient alternatives to standard RNNs, such as minimal RNN [31], minimal gated unit (MGU) [229], MGU1, MGU2, MGU3 [84]. The MGUs differ from GRUs by reusing the gates, removing the bias term or the weight matrix completely, or a combination, detailed as follows:

$$\begin{aligned} &\text{MGU1:} \quad f_t = \phi\left(U_f h_{t-1} + b_f\right), \\ &\text{MGU2:} \quad f_t = \phi\left(U_f h_{t-1}\right), \\ &\text{MGU3:} \quad f_t = \phi\left(b_f\right), \end{aligned} \tag{4}$$

where f_t is the unique gate of the recurrent unit with weight parameters U_f, bias b_f, and h_{t-1} the previous hidden state. We notice that the MGU1, MGU2, and MGU3 variants do not directly gate the current input x_t, but instead, they indirectly gate the previous input x_{t-1} by gating the previous state h_{t-1}, that has processed the previous input x_{t-1}. Heck and Salem [84], Zhou et al. [229] suggest that these alternatives are competitive with GRU in terms of accuracy with a smaller parameter budget and thus should be more low-power friendly.

Next, we explore the methods that apply directly to models in order to reduce their power footprints.

Model compression techniques. Model compression is a set of methods aiming to address the growing power footprint and costs associated with the deployment of neural networks in terms of size and computation on resource-constrained devices [87, 152], such as MCUs. In the following sections, we will provide an overview of the most commonly used techniques, which essentially encompass five methods: knowledge distillation, pruning, quantization, weight-sharing, and low-rank matrix decomposition [152].

4.1 Knowledge distillation.

Knowledge distillation is a high-level approach to model compression, first explored in Buciluǎ et al. [26] to reduce the model size by learning a small (student) model from an ensemble of models (teacher). Then Hinton et al. [85] popularized knowledge distillation for neural networks where a small model (student) is trained from the supervision of a larger and overparameterized trained model (teacher) that has learned "dark knowledge". The idea is to leverage the latent knowledge the large teacher has captured and transfer it to the student during the training process. The loss encompasses both the original student loss (e.g., cross-entropy) and the difference

between the teacher and student distribution, expressed as follows:

$$L_{\mathrm{KD}}(x, y) = \alpha L_{\mathrm{S}}(x, y) + (1 - \alpha)\mathrm{D}_{\mathrm{KL}}\left(\mathrm{softmax}\left(\frac{T(x, y)}{\mathrm{temp}}\right),\ \mathrm{softmax}\left(\frac{S(x, y)}{\mathrm{temp}}\right)\right), \tag{5}$$

where L_S is the student loss function, $S(x, y)$ is the output of the student model, $T(x, y)$ is the output of the teacher model, D_{KL} is the Kullback–Leibler (KL) divergence, $\alpha \in [0, 1]$ is a hyperparameter that controls the amount of distillation given by the teacher to the student, and temp is another hyperparameter that softens the probability distributions of the output models.

In practice, we must choose and train one teacher and one student architecture. Hinton et al. [85] showed promising results across general computer vision tasks and sequential data. However, the disadvantages are that it requires empirical knowledge to find good teacher and student models, as well as additional computations to train the teacher and the forward pass of the teacher during the student's training. Although the design of the teacher would consist of training an overparameterized model, which works well in practice, the student should be the size of our target model. Moreover, we can bypass the additional forward pass of the teacher by storing its output along with the training set.

Therefore, the general framework design of knowledge distillation is flexible for our case and has proven promising performance in a wide range of applications.

4.2 Model pruning

While knowledge distillation involves training a new smaller model, pruning focuses on removing less important parts of a model. From a neuroscience perspective, the human brain has a pruning mechanism that removes redundant connections or irrelevant information from past experiences [152, 210]. In the case of deep learning models, they are notoriously overparameterized (Section 2.2), which provides them with a large degree of freedom. In fact, it has been found that only a small fraction of the total parameters are critical [45]. Model pruning is a very active research area at the intersection of promoting efficient deep learning and understanding neural network training and generalization ability, where new methods emerge continuously [4, 54, 87]. Recent comprehensive surveys classify and discuss a large number of pruning strategies, including [33, 82, 202]. In contrast, our review narrows the discussion to pruning approaches that have demonstrated feasibility for ultra-low-power inference on microcontrollers, highlighting implementation aspects, dataset scale, and compatibility with current TinyML toolchains such as CMSIS-NN [112] and TensorFlow Lite Micro [44, 212].

Han et al. [78], Ullrich et al. [200] made a major breakthrough for model compression in the modern deep learning era, where they combined pruning, quantization (Section 4.3) and Huffman encoding [93] to reduce a CNN model by 49 times its size with less than 0.5% accuracy loss on the ImageNet dataset.

Seminal work by Frankle and Carbin [53], Liu et al. [134] provided more theoretical understanding; the lottery ticket hypothesis (LTH) states that there exists a sparse subnetwork (winning ticket) that can be trained from scratch with the same initialized weights and reach the performance of the original network (10 times larger). In this view, a large model has a greater chance of containing a good subnetwork. They suggest that the network architecture itself is more critical than keeping the values of the weights in the original trained network. In practice, Frankle and Carbin [53] requires iterative pruning trials of subnetworks to find the winning ticket, which is computationally expensive. Further work extended the LTH, showing that universal tickets could be reused across other applications [27, 52]. In particular, Ramanujan et al. [165] generalized the LTH to the strong lottery ticket hypothesis (SLTH) where the subnetwork performs well with the randomly initialized parameters and thus does not require retraining. Additionally, Burkholz et al. [27] demonstrates that SLTH can also yield universal tickets across other applications. Consequently, the SLTH promises that training deep learning models could be replaced by efficient neural network pruning [52]. Alternatively, pruning can be seen as a form of neural architecture search (NAS) [48], aiming to find Pareto-optimal architectures [134]. Moreover, it is also a

form of regularization because it reduces the complexity of the model, similar to dropout, but the effect remains permanent.

There are essentially two types of pruning: unstructured and structured pruning, referring to how the pruning is performed in a weight matrix of a model, as illustrated in Figure 5.

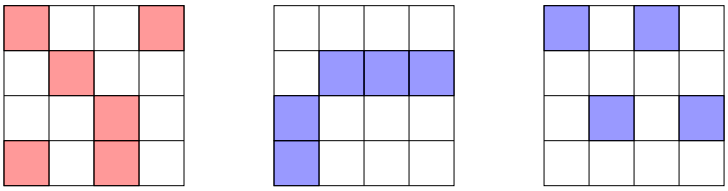


Fig. 5. Unstructured pruning (left panel) versus structured pruning (middle and right panels).

Unstructured pruning. Unstructured pruning refers to the removal of fine-grained weights in contrast to a group of weights. It is the simplest and most sparsity-inducing type of pruning because trained neural networks are less sensitive to one weight than a specific block.

The most intuitive pruning scheme is to remove weights based on their absolute values, which is the simplest form of magnitude-based pruning, so it does not require any data. This simple approach has been studied early [76] and is very effective [58, 78, 87, 231]. In general, it involves re-training to adapt the model to its new architecture.

While there are a plethora of pruning algorithms, Gale et al. [58] suggested that magnitude-based pruning provides state-of-the-art or comparable performance to other pruning methods [138, 196].

In particular, Zhu and Gupta [231] introduced a gradual sparsity technique using a polynomial during the training schedule as follows

$$s_t = s_f + (s_0 - s_f)\left(1 - \frac{t - t_0}{n\Delta t}\right)^3, \tag{6}$$

for $t \in \{t_0, t_0 + \Delta t, \dots, t_0 + n\Delta t\}$, where s_t and t are the current sparsity and step, s_f is the target sparsity, s_0 and t_0 are the initial sparsity and training step (usually 0), n is the number of pruning steps, and Δt is the pruning step frequency. In other words, at every Δt, a gradual number of weights is set to zero based on their magnitude until we reach the desired sparsity level. The objective of the polynomial schedule is to prune quickly and early when there is the most redundancy, and then slow down the pruning rate as there is little remaining redundancy [231].

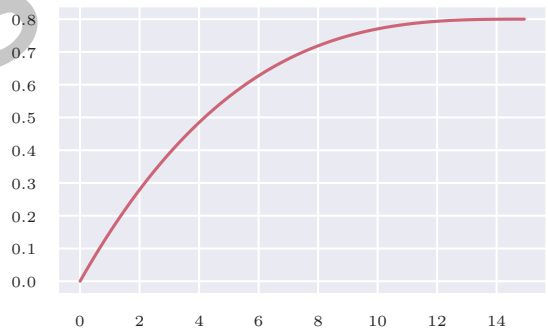


Fig. 6. Pruning rate over epochs with a polynomial schedule function [231] with $s_f = 0.8$, $s_0 = 0$, $t_0 = 0$, $n = 13260$, $\Delta t = 100$ (Equation (6)).

Noting that pruning is a form of regularization, Golatkar et al. [66] found that the early regularization phase is the most critical to performance and that late regularization can even worsen the results, thus supporting the effectiveness of polynomial schedule.

The advantage of magnitude-based pruning is that it is model- and task-agnostic, can seamlessly incorporate within training, and is easy to implement. Moreover, progressive pruning [231] is natively supported by the TensorFlow framework [1]. Additionally, they demonstrate a 90% sparsity rate with acceptable accuracy loss and found that their approach on large-sparse networks performs better than their smaller-dense counterpart. An explanation of this is that larger models are easier to prune because the magnitude of single weights becomes smaller as the model grows larger when the model has converged [152]. However, the biggest disadvantage is that unstructured pruning results in sporadically induced weights, which may be difficult to efficiently leverage on embedded hardware, but previous work demonstrated that it is possible to leverage high sparsity with practical encoding [78].

Structured pruning. Structured pruning alters the architecture of the neural network in blocks, such as neurons, filters, or an entire row or column of a weight matrix. Structured pruning can be induced by using a systematic criterion based on redundancy, as in Srinivas and Babu [188], where neurons were removed in neural networks by identifying duplicate pairs of neurons, performing a recovery step to compensate for removal. Another common approach is to use regularization penalty to encourage pruning at the channel level in CNN models [83, 133], by neurons [5], or layers [213], resulting in models with 60% sparsity without significant loss. The clear advantage of structured pruning is that it is hardware efficient because it may allow skipping entire filters or rows during a matrix multiplication, as suggested in Figure 5. However, block-based pruning techniques have strict compression rules that make them more difficult to achieve without degrading performance and require a certain amount of block sparsity to obtain a faster run time than baseline [196]. However, recent research suggests that wider and sparser networks generalize better than their smaller dense counterparts designed by structured pruning [12, 67, 123, 199, 231].

Pruning based on Bayesian methods. Among all, Bayesian inference can be used to promote sparsity in the model. Bayesian methods provide the posterior distribution over the parameters of the model, given the dataset and a prior distribution. As a result, this posterior distribution encompasses more information than a simple vector of optimal parameters: variance of the parameters, thickness of their tails, etc. Besides, by tuning the prior distribution, the user can impose some structure to the posterior distribution, which can be used to encourage sparsity in the model.

A popular and intuitive prior is the *spike-and-slab* prior, introduced by [148] and used in neural networks by [189], for instance. This prior is a mixture between a Dirac at 0 (the *spike*) and a distribution with a continuous density (the *slab*), e.g., a zero-mean Gaussian distribution:

$$p(x) = p_0\delta(x) + (1 - p_0)(2\pi\sigma_0^2)^{-1/2} \exp\left(-x^2/(2\sigma_0^2)\right),$$

with $p_0 \in (0, 1),\ \sigma_0 > 0$. That way, the spike-and-slab prior pushes the parameters towards 0. More complex, the *Horseshoe* prior [30, 65] has been designed to have an infinite density at 0 and Cauchy-like tails:

$$\begin{aligned} X_i \,|\, \lambda_i, \tau &\sim \mathcal{N}(0, \lambda_i^2\tau^2), \\ \lambda_i &\sim \mathcal{C}^+(0, a), \\ \tau &\sim \mathcal{C}^+(0, b), \end{aligned}$$

with $a > 0,\ b > 0$, and where $\mathcal{C}^+(0, a)$ is the half-Cauchy distribution with scale parameter a. Thus, the horseshoe prior encourages the parameters to be exactly 0 while allowing extreme values. Another regularization technique, the drop-out [191], has led to the development of the *log-uniform* prior by [153]. Although improper, this prior is

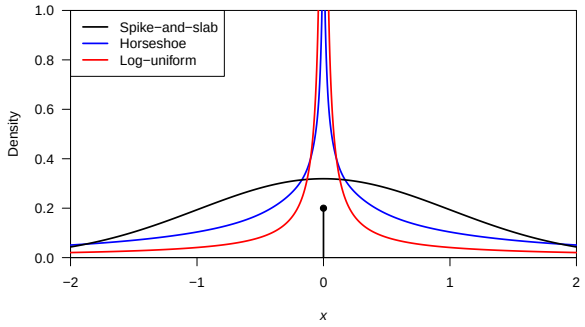


Fig. 7. Prior densities promoting sparsity illustrated with the following hyperparameters: Spike-and-slab with $p_0 = 0.2$ and $\sigma_0 = 1$; Horseshoe with $a = b = 1$; Proper log-uniform with $a = 10^{-5}$ and $b = 2$.

designed to be agnostic about the order of magnitude of the parameters. As a result, its density tends to infinity at 0, so small values are encouraged. However, to make the log-uniform prior proper, it is common to set its density to 0 outside an interval spanning several orders of magnitude, as described follows:

$$p(x) = (2|x| \log(b/a))^{-1} 1_{[a,b]}(|x|),$$

with $0 < a < b$. These densities are illustrated in Figure 7.

Beyond the choice of the prior, one should pay attention to the choice of the *approximate* Bayesian method and the search space of the approximate posterior. In fact, it is usually too costly to compute the exact posterior distribution of the parameters of large models such as neural networks [7, 159]. Therefore, one has to choose an approximate Bayesian method and a search space of the posterior distribution. For instance, it is common to use *variational inference* [71] and look for an approximate posterior consisting of independent Gaussian distributions over the set of parameters (where their mean and variance are trained). In [189], the candidate posterior distributions for one parameter θ are the mixtures between the Dirac at 0 and $\mathcal{N}(\mu, \sigma^2)$, with mixture parameter g: the trained parameters are then g, μ, σ. In that case, the value of g is directly related to the sparsity: if $g = 0$, then $\theta = 0$, so θ can be pruned.

Summary. In summary, pruning has strong theoretical and practical incentives that make it a high-potential and relevant choice. Unstructured pruning approaches are more flexible across diverse architectures and yield the highest sparsity rate, while structured pruning approaches are more hardware efficient.

Moreover, multiple works have shown that combining pruning with other model compression methods, such as quantization, can produce a high compression rate without significant performance loss [78, 203, 226].

4.3 Quantization

A different perspective on model compression is quantization. It is a method mapping input values from a larger set (often continuous) to a smaller set (often discrete) [64] to find lossless approximations of numerical input values, and can be seen in related work dating back to the 1800s in the foundations of calculus (e.g., least-squares, approximation of integrals) [64, 73].

Specifically, fixed-point attempts to represent continuous values (larger set) with a fixed amount of precision (smaller set); thus, quantization is a mandatory method to meet the low-power requirements of fixed-point arithmetic inference on MCUs, as stated in Section 3.3.

Recent work on neural network quantization builds upon prior work but presents unique challenges due to the high power footprint and overparameterized nature of deep learning models. The inherent redundancy in deep

learning models allows for some leniency in quantization errors, limiting accuracy loss [64, 74]. Consequently, very small models, that can be found in TinyML, should be more sensitive to quantization.

Minimizing quantization performance loss can be seen as an optimization problem, where the objective is to find a discrete distribution (quantized weights) that is closest to the original distribution (real weights, activation, or data). In practice, this translates by rounding or truncating the model's parameters (weights, activations) and data from floating points (e.g., 32-bits) to integer values (e.g., 8-bits).

Compared to pruning, quantization often results in less accuracy loss because weights lose precision but are not removed, hence a lower level of information loss [173]. For reviews of quantization techniques, one can refer to Liang et al. [124], Rokh et al. [169]. Unlike these works, which focus on general-purpose or mobile-level quantization methods, our review specifically addresses quantization under the extreme memory and compute constraints of sub-100 kB microcontrollers, emphasizing practical deployment trade-offs and framework-level support within the TinyML ecosystem.

Quantization approaches can be characterized by several factors: the stage of the quantization process as quantization-aware training (QAT) or post-training quantization (PTQ), the type of quantization steps as uniform or non-uniform, and the arrangement of quantization levels around the zero-point Z as symmetric or asymmetric (Equation (7)).

Quantization-aware training (QAT). QAT involves integrating quantization into the training process or fine-tuning the model by simulating the effects of quantization during the forward or backward pass. However, the quantized function is not differentiable (Equation (7)) and can result in zero-gradients in low bit-precisions, making it difficult to train the model. Prior works have quantized values in the forward pass and used real values during the backward pass such as the straight-through estimator (STE) [18, 41, 97], or other approximations [136, 220]. In addition, Choi et al. [35] learns to optimize the range of activation clipping values and then linearly quantize both weights and activations to 4-bits, while Bhalgat et al. [20] uses a gradient estimate to learn scaling factors of weights and activations. Alternatively, Darabi et al. [43] employ regularization to force the weights to converge to binary values during training, which is generalized in Lê et al. [115] to any bit-precision and using a schedule for progressive quantization during training. The objective of QAT is to obtain a stabilized quantized model by the end of training. These methods enable below 8-bit quantization and even down to 1-bit weights or activations [41, 92, 135, 164, 166] with competitive results compared to full precision networks and PTQ. Additionally, AskariHemmat et al. [9] found that quantization is a form of regularization, where the induced quantization noise can help improve generalization, and particularly to 8-bits on several computer vision tasks. However, QAT often requires a lot of tuning, additional computation, and access to the dataset to re-train the model, especially for low-bit quantization.

Post-training quantization (PTQ). PTQ is the simplest and fastest approach, where quantization can be applied to any trained model without re-training or access to the dataset [15, 29, 38, 49, 78]. Previous work corrected the mean and variance of quantized weights [15, 64], or minimized the mean squared error between the quantized and full-precision distributions [38], allowing 4-bit quantization with acceptable performance. Another approach used piecewise linear functions to partition the quantization range into non-overlapping regions for each weight in order to minimize the quantization error [49].

The most widely used quantization method for MCUs is uniform affine PTQ to int8 because it is straightforward and supported by MCUs [64, 107, 173]. Moreover, uniform PTQ with int8 provides sufficient performance compared to the original full-precision 32-bit (FP32) model for a wide variety of NNs [49, 107, 119]. However, PTQ may lead to a more significant loss in accuracy, especially for quantization below 8 bits [15, 64].

(Non-)uniform quantization. In uniform quantization, the quantization steps are evenly spaced, so it is the most straightforward type of quantization while being natively supported in all embedded hardware [173].

In contrast, non-uniform quantization may better capture the original distribution, thus yielding higher accuracy [64]. For example, Miyashita et al. [149], Zhou et al. [228] uses a logarithmic distribution with exponential quantized steps instead of linear steps. Alternatively, Fu et al. [55] quantize activations and gradients by finding optimal quantization points that fit their full-precision distributions based on their Weibull prior properties [208, 209], and obtained competitive results compared to the full precision training using less bits than their uniform-based counterpart [55].

However, non-uniform quantization schemes are challenging to deploy on embedded hardware because they require a custom implementation to efficiently exploit their specific distribution, in contrast to uniform quantization which is deployable out of the box. Therefore, we restrict the scope of our review to uniform quantization schemes for a wide hardware support.

(A-)symmetric quantization. In symmetric quantization, the lower and upper bounds of the quantization range are equidistant from the zero-point, and $Z = 0$, which simplifies as follows

$$Q(r) = \text{int}(r/S) - Z, \tag{7}$$

where Q is the quantization function, r the value to quantize, S a scaling factor, Z represents the zero-point value in the integer discrete space, α, β denote the lower and upper bounds ($\alpha < \beta$), respectively, of the clipping range where we constrain r, and b is the bit-width.

The scaling factor for symmetric and asymmetric quantization is computed as follows:

$$S_{\text{sym}} = \frac{\max(|\alpha|, |\beta|)}{2^{b-1} - 1}, \quad S_{\text{asym}} = \frac{\max(|\alpha|, |\beta|)}{\left(2^b - 1\right)/2}. \tag{8}$$

Asymmetric quantization schemes consider the full range of quantized values, e.g., $[-128, +127]$, in contrast to $[-127, +127]$. This provides a slightly larger range to minimize quantization error but is a more complicated implementation due to the zero point $Z \neq 0$ in Equation (7), and may lead to more computational overhead [215].

Quantization based on Bayesian methods. Similarly to the case of pruning, Bayesian inference can be used to reduce the number of bits necessary to encode a continuous parameter. For instance, [203] have proposed a method in the variational inference framework [71]: each parameter of a neural network is decomposed as a sum of gated residuals:

$$x = z_2(x_2 + z_4(\epsilon_4 + z_8(\epsilon_8 + z_{16}(\epsilon_{16} + z_{32}\epsilon_{32})))),$$

where x_2 is the basic 2-bits approximation of x, the ϵ_n are the n-bits residuals of x, and the z_i are the corresponding gates. In this example, x is allowed to be pruned or approximated on 2^n-bits for $n \in \{1, 2, 3, 4, 5\}$. The $(z_i)_i$ are dependent Bernoulli random variables whose parameters are trained: if all z_i tend to become 0, then x can be pruned; if z_2 tend to be always 1 and the others 0, then x can be efficiently approximated by its 2-bits part; if all z_i tend to be always 1, then x should remain coded on 32 bits. In this setup, the optimal level of quantization (in a Bayesian sense) is discovered progressively during training and can be heterogeneous across the parameters. Moreover, the allowed quantization levels span a large interval, from the usual 32-bits quantization to pruning.

Also, the entire posterior distribution provided by Bayesian inference can be used to improve quantization methods. For instance, [219] have developed a quantization method that can be applied to a model for which a posterior distribution is already known for each of its parameters. In this work, the posterior distribution of each parameter is transformed by a function, which is the CDF of the prior distribution. Then, the mode of the resulting function is quantized with precision depending on its width: if the mode has a large width, then a few bits are necessary to encode it. With this setup, the partition used for quantization is, at least, adapted to the prior distribution, and leads to a more efficient quantization when applied to posterior distributions. Finally, [146]

trains binary neural networks using the Bayesian learning rule [104], an algorithm inspired by the Bayesian paradigm. This approach enables uncertainty quantification while providing state-of-the-art results.

Summary. In summary, quantization methods have a long history and exist in many flavours to achieve lossless approximations in the most constrained settings. QAT emerges as a superior option in below 8-bit settings, but is more complex and requires more computations than PTQ.

However, uniform PTQ with lower bit quantization is more sensitive due to the distributional properties of weight, which are clustered around zero (Gaussian or Laplacian) [78, 128], and few of them are in a long tail (Sub-Weibull) [208, 209]. Consequently, uniform quantization maps too few quantization levels to small weights and too many to large ones, leading to performance loss [49]. However, overparameterized models are less sensitive to PTQ due to having more degrees of freedom [152] in contrast to smaller models. Thus, we would favour uniform 8-bit PTQ due to its simplicity and acceptable results until we need lower-bit precision for more power footprint reduction.

4.4 Weight-sharing

Weight-sharing is the simplest form of model compression, involving sharing weights values in different parts of the model, so it imposes a model architecture prior to training [152]. We could set the amount and location of weight-sharing in a strategic way in the model, such as in rows or columns of the weight matrix, for efficient inference. However, manual weight-sharing design may be difficult because we cannot predict the final performance, even if redundancy is part of the design of deep learning models.

Prior works have used an automated approach, such as clustering weights with K-means that shares the centroid value among weight clusters with re-training [216], where they compressed a CNN model by a factor of three without significant loss, or by using a penalty term to encourage grouping weight [156, 200].

In particular, quantization is a form of weight-sharing because lowering the bit-precision of parameters forces them to be aggregated into a common set of values.

4.5 Low-rank matrix and tensor decompositions

Since neural network weight parameters are essentially matrix or tensors, we can apply approximation methods from linear algebra such as single value decomposition (SVD) or its generalization to tensor decomposition (TD) [152]. The weight matrix is then replaced by a product of two lower-rank matrices [6, 155, 174, 217]. In particular, Alvarez and Salzmann [6] obtained a compression rate of up to 96% compared to the original model.

However, these methods require additional hyperparameter tuning [116], as well as trial and error to find the optimal rank, which may not generalize between applications. Furthermore, for MCUs, it is crucial to consider that the incorporation of additional products from the lower rank matrix may not always lead to increased efficiency and reduced power consumption, so further evaluation of the device is required.

4.6 Summary

In summary, we have provided a comprehensive overview of the key methods to design and train efficient TinyML models, accompanied by their related theoretical concepts and practical implications. These methods have generated growing interest, as they bridge the gap between deep learning theory and the deployment of efficient neural networks.

Specifically, model pruning, knowledge distillation, and quantization have demonstrated very promising compression rates, particularly in larger-scaled networks (Mobile or Cloud size) that are more robust to model adjustments. Furthermore, some model compression methods are also forms of regularization that can even help the model to generalize better. Thus, these approaches show high potential to meet the ultra-low-power requirements MCUs.

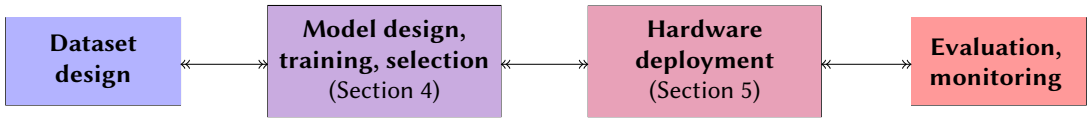


Fig. 8. TinyMLOps pipeline.

In practice, since TinyML is at an early stage, tools and processes are not mature enough yet to evaluate and truly leverage the high compression rate of existing methods for ultra-low-power MCUs, so we will review practical TinyML tools and aspects of the deployment of compressed neural networks in the next section.

5 Deploying deep learning models on ultra-low-power MCUs

In this section, we define and review existing tools and methods for the end-to-end deployment of efficient neural networks on ultra-low-power MCUs.

TinyMLOps. The first framework for training deep learning models was developed in 2008 [198], with TensorFlow [1] and PyTorch [160] following suit in 2015 and 2016, respectively. These frameworks enabled the large-scale development and deployment of deep learning models, which in turn led to the emergence of Machine Learning Operations (MLOps) [106]. MLOps consolidates best practices and outlines steps for mitigating technical debt [180] during the development and deployment of machine learning systems.

In contrast, the earliest known publication on TinyML dates back to 2019 [77], and the first dedicated deep learning framework for microcontrollers, TensorFlow Lite for Microcontrollers (TFLM), was also released in 2019 [44, 212]. As TinyML gained traction in the industry, MLOps naturally expanded to include TinyMLOps as a subset [114, 120, 172], focusing on refining the process of deploying machine learning on embedded devices, as depicted in Figure 8. In the context of TinyML, deployment refers to the process of taking a trained model and enabling it to run on an embedded system, such as compiling the model, firmware integration, and verification of the solution on the target device.

Consequently, the TinyMLOps ecosystem is still in an earlier stage than MLOps, with challenges yet to be fully addressed. We detail here the challenges faced by TinyMLOps tools and methods in practice, as well as existing solutions.

5.1 Efficient methods and deployment

In the context of edge inference, quantization is a mandatory step for all methods to ensure efficient deployment on resource-constrained devices. Consequently, the specific details of quantization are not reported separately here. This section reviews the following model compression techniques in practice, echoing their presentation in Section 4: knowledge distillation, pruning, weight sharing, and low-rank matrix and tensor decomposition, which are employed to create efficient TinyML models suitable for edge devices.

Knowledge distillation. TinyBERT [99] utilized knowledge distillation on a large language model, resulting in a model that is 7.5 times smaller and 9.4 times faster. However, it remains too large for microcontroller deployment with 14.5 million parameters. [163] trained a quantized student model from a full precision teacher, producing a small student model with 77.92% accuracy and 450KB memory usage on CIFAR-10, making it 46 times smaller than the teacher. Using a medium-sized student model, they achieved 84.22% accuracy, though it was almost three times the size of the small student model. [224] combined standard knowledge distillation with post-training quantization, achieving 69.5% accuracy and an 81KB model size on CIFAR-10.

Despite these promising results, there is limited use of knowledge distillation for deployment on microcontroller units (MCUs) in the existing literature. This may be attributed to the simplicity of pruning and quantization methods and the more stringent size constraints compared to mobile-sized models (approximately less than 1MB).

Pruning. In practice, all structured pruning methods can be applied prior to deployment, benefiting efficient TinyML models. Consequently, many previous works already utilize structured pruning as an effective compression method, which can be well adapted to the TinyML context.

Structured pruning, as demonstrated in [50] with Bayesian compression [137] via variational inference, approximates the weight posterior by a certain distribution. They achieved an 80-fold reduction in parameter count while maintaining an accuracy of 98.64%, resulting in a 2.77KB model using 1.96KB RAM on MNIST. Similarly, [127] used structured pruning to achieve 91.98% accuracy on CIFAR-10 with a 256KB model, and up to 96.03% accuracy on Google Speech Commands v2-12 with models as large as 115KB.

While structured pruning is effective out of the box, unstructured pruning poses challenges in fully leveraging its benefits for edge inference, making it an area for future development.

Weight sharing. Weight sharing can be seen as a loose form of quantization since parameters are approximated into a finite set of values and thus are merged into the same quantized value. [78] combined pruning, weight sharing, and quantization, achieving 98.42% accuracy on MNIST with a model size of 27KB, compared to the original 1070KB model. Though weight sharing shows potential for efficient model deployment it has not been widely adopted for TinyML models.

Low rank matrix and tensor decomposition. [111] combined sparse matrix decomposition and quantization, achieving 92.21% accuracy on Google Speech Commands v2-12 with a model size of 57KB. Despite being less straightforward than other approaches, these methods are under exploration for efficient inference.

5.2 Challenges for TinyML tools

The fundamental characteristic of TinyML is the tight dependency between software and hardware components. In fact, failure to adapt the delivered machine learning software to the constraints of particular hardware renders it unusable, resulting in wasted efforts in previous TinyMLOps steps. Additionally, the diverse landscape of embedded hardware further complicates the task of developing a versatile software base capable of supporting a wide range of embedded hardware platforms [120, 172], resulting in a manual and iterative approach to the design of new models. As a result, designing new models that work on different hardware remains a manual and iterative approach (different firmware, debugging interfaces…). The challenge of TinyMLOps is to improve the entire pipeline, from design to deployment, from data to computation.

Even though TinyML shares some tools with traditional ML (e.g., TensorFlow, PyTorch, Tensorboard), its more recent emergence means that specialized tools are not yet created or are less mature in providing comprehensive solutions. As the TinyML community continues to grow, greater awareness and adoption of tools will lead to faster innovation and the development of comprehensive solutions.

5.3 TinyML tools

We restrict TinyML frameworks to the one that supports TensorFlow models as input due to its wide adoption in the industry and that also targets Arm Cortex-M MCUs for inference.

We essentially consider these two common approaches to TinyML frameworks [185]:

(1) Using a runtime that loads the model from read-only device memory at runtime (e.g., TensorFlow Lite Micro),
(2) Using a transcompiler that converts and compiles models to C or C++ code that then can be built within a project (NNoM, Edge Impulse, μTVM).

5.3.1 Low-level library.

CMSIS-NN. CMSIS-NN (Cortex Microcontroller Software Interface Standard for Neural Networks) is a low-level library specifically developed by Arm [112] for the Cortex-M microcontroller ecosystem (Table 4). It provides a collection of efficient neural network core functions for low-level acceleration. These functions include optimized operations for common neural network operations, such as fully-connected (FC) layers, convolutions, and activation functions (ReLU, sigmoid, tanh…). CMSIS-NN has been shown to provide a 4.6x speedup and 4.9x energy savings over non-optimized convolutional models [112, 173].

5.3.2 TinyML frameworks.

TensorFlow Lite Micro (TFLM). This framework is an extension of the TensorFlow ecosystem, specifically designed for deploying neural networks on low-power MCUs such as ARM Cortex-M [44, 167, 173, 185, 212]. TFLM emphasizes portability by discarding uncommon features, data types, and operations and avoids reliance on specialized libraries or operating systems, thereby achieving memory efficiency and support for a wide range of hardware. It converts and quantizes a 32-bit floating-point TensorFlow model to a compressed flat buffer file (.tflite) using 8-bit integers for weights and 32-bit integers for activations and data. TFLM uses an interpreter-based approach to process the neural network graph at runtime and consists of three primary components: operator resolver, memory stack pre-allocation, and interpreter [179, 187]. The operator resolver links only essential operations to the model binary file, and the memory stack is used for initialization and storing runtime variables. The interpreter resolves the network graph at runtime, allocates the memory stack, and performs runtime calculations. More technical details are provided in David et al. [44], Schizas et al. [179].

However, TFLM has limitations, such as missing support of some layers or operations (GRU, Conv1D, some important activation functions…), arbitrary bit-widths of weights, and activations. Moreover, TFLM lacks target-specific optimizations during compilation because it relies on a graph-level representation that does not include device-specific function kernels and execution details [179, 187], and can result in larger memory usage, so it may not meet our extreme memory requirements. Moreover, it does not provide built-in tools to measure power footprint metrics such as inference time or memory usage. Moreover, the interpreter-based approach at runtime makes it difficult to debug and extend, compared to standard compiled code, which hinders research efforts. Despite these limitations, TFLM remains the most popular choice for microcontroller-based deep learning applications.

Neural Network on Microcontroller (NNoM). This open-source framework [141] relies on a C code generation approach with a set of function calls. It is flexible, easy to debug, and supports a wide range of MCUs, but only supports models created using TensorFlow. The project includes a compiler that converts and quantizes a TensorFlow model to plain C code with 8-bit weights and 32-bit activations and data. Additionally, the NNoM compiler supports the CMSIS-NN to generate optimized code for ARM Cortex-M processors [185]. It does support all RNN layers including GRU, in contrast to TensorFlow. However, it does not support lower bit-width quantization and has a smaller community and adoption compared to TFLM, so this hinders the development of new features.

Edge Impulse. Lastly, Edge Impulse [98] is a closed-source cloud service that develops TinyML machine learning models for edge devices and supports AutoML for mobile and microcontrollers [167, 173]. Edge Impulse provides a complete end-to-end model deployment solution, including data collection, feature extraction, training, and deployment [173], with an intuitive graphical interface and a friendly no-code approach. The training is carried out in the cloud and the learned model can be exported to an edge device using a data-forwarding capable connection [179].

For model deployment, Edge Impulse uses an interpreter-less edge-optimized neural compiler, which directly compiles the model into C++ source code. This approach eliminates the need to store unused ML operators, resulting in reduced memory requirements at the expense of portability compared to TFLM. Studies have shown that the EON compiler can run the same model with 25%-55% less SRAM and 35% less flash memory than TFLM [173].

In conclusion, TinyML brings together the embedded systems and machine learning communities, which have traditionally operated independently. Both academia and industry have developed several software frameworks for TinyML to streamline the deployment of machine learning models on microcontrollers. In particular, we are interested in TFLM because it integrates with TensorFlow and provides a complete toolchain for deploying low-power models MCUs. We are also interested in NNoM because it provides a flexible and simple approach to quantizing and deploying models from plain C code and CMSIS-NN support for Arm Cortex-M MCUs. Moreover, these two frameworks are open-source, which makes them accessible as well as potentially extendable. However, these frameworks are still in the early stages of development, with some missing features and functionality. Despite their limitations, the current first generation of TinyML tools can transition the state-of-the-art machine learning models to ultra-low-power environments.

5.4 Algorithm-hardware co-design

Even though the diversity of TinyML hardware makes designing new models difficult, working at a low level allows us to design new processors adapted to specific tasks. A complete algorithm-hardware co-design workflow with extension of the Instruction Set Architecture (ISA) RISC-V has been proposed by Verma et al. [206]. It involves all the stages between hardware design and commonly used ML libraries such as PyTorch or TensorFlow. Specifically, on the software side, they use a compiler to translate into C the functions defined in ML libraries, and, on the hardware side, they use the generated C code to design a processor, along with a Software Development Kit (SDK) and a specific set of instructions. As a result, they obtained a 17.63× speed-up for the general vector-matrix multiplication (GEMV) kernel by using a 16×16 custom Vector–Matrix Multiply (VMM) instruction and specific functional hardware.

For more details about hardware and software acceleration in IoT, one can refer to [2, 121].

5.5 Experimental results

An important benchmark involving TinyML has been proposed by Reddi et al. [168]. Its purpose is to evaluate the latency and energy when performing inference on one single input, for a given model on a given MCU. The benchmark consists of 5 different combinations of tasks/models, which represent typical usages of TinyML:

- keyword spotting: Speech Commands dataset [211], Depth-Separable CNN model [36];
- anomaly detection: ADMOS Toy Car dataset [105], Fully Connected AutoEncoder;
- person detection: COCO (visual wakeword) dataset [39], MobileNetV1 (0.25x) [90];
- image classification: CIFAR-10 dataset [108], ResNet-V1 model [81].

The latency and energy consumption are measured after checking that the tested model meets a certain level of accuracy, determined by the benchmark. For instance, the ResNet-V1 model must achieve at least 85 % accuracy on CIFAR-10.

A rapid comparison of the Cortex M4 and M7 in Table 6 shows that, for a trained model, the energy consumption of the M4 is lower than that of the M7, while having a higher latency. These experimental results are consistent with Table 4, showing that the Cortex-M7 has both a higher frequency and consumption than the Cortex-M4.

While benchmarking efforts such as MLPerf Tiny have standardized latency and energy evaluation, other practical aspects of deployment remain less explored. In particular, the design of lightweight *preprocessing pipelines* on-device, covering tasks such as denoising, normalization, and fixed-point feature extraction, plays a

Processor	DS-CNN		FC AutoEncoder		MobileNetV1 (0.25x)		ResNet-V1	
32-Bit ARM	lat.	ener.	lat.	ener.	lat.	ener.	lat.	ener.
Cortex-M4	88.36	1376	42.01	2965	816	165.7	24.48	5090
Cortex-M7	23.16	1820	11.25	3895	190.1	203.37	6.06	6919

Table 6. Latency (in ms) and energy (in µJ) of each model at inference on two MCUs for a single input. The models have been previously trained on specific tasks to meet a given level of accuracy. Reported metrics have been measured on the version 1.2 of the benchmark. Full results have been published by ML Commons (https://mlcommons.org/benchmarks/inference-tiny/).

crucial role in ensuring reliable model performance under tight memory and latency budgets. These preprocessing steps are often performed offline or are inconsistently implemented, making reproducibility across platforms difficult. Furthermore, TinyML evaluation metrics should extend beyond accuracy to incorporate end-to-end latency, memory footprint, energy per inference, and model size, reflecting the actual constraints of embedded devices. Developing unified, open benchmarking methodologies that integrate these dimensions would enable consistent cross-platform comparison and guide future optimization of both models and hardware for real-world TinyML applications.

6 Limitations of TinyML

In this section, we assess the limitations of current TinyML models when applied to standard datasets, with a specific focus on their memory size. The emphasis is on memory size rather than other metrics such as latency, as memory size is the primary constraint to overcome for deploying TinyML models. Latency is considered as secondary and is less frequently reported in the literature. Consequently, memory size is crucial in determining the range of microcontrollers that can be selected, which is critical for industrial applications. Our primary objective is to identify the most efficient models that strike the optimal balance between performance and memory usage. By doing so, we aim to offer valuable insights to researchers and industry professionals, shedding light on the scale of TinyML models. Furthermore, our analysis aims to pinpoint the most suitable models among widely adopted options and various hardware platforms for their respective applications.

We focus here on the most common datasets found in TinyML model benchmarks for the following three tasks:

- *Image Classification*: **MNIST** is a basic dataset for image classification of handwritten digits. We also use **ImageNet**, a more challenging image classification dataset than MNIST due to its larger and more diverse images and labels, thus requiring more complex models.
- *Image Recognition*: **Visual Wake Word** is focused on the visual presence recognition of a person or an object in images.
- *Speech Recognition*: **Google Speech Commands v2-12** consists of short audio clips of spoken word commands with 12 classes to recognize.

MNIST. For MNIST, we find that μNAS [125] clearly offers the best size-accuracy tradeoff and is below the Cortex M0+ memory limitation. The large LeNet [78] has slightly better accuracy but is over the memory threshold. Then, the two versions of Sparse CNN [50] are both below the extreme low-power threshold, but their accuracies are still lower than μNAS. However, ProtoNN [75] and Bonsai [110] display the least favorable tradeoff, but ProtoNN is below the Cortex M0+ threshold.

ImageNet. We observe that ImageNet models require the largest models of all studied here, mostly above the ultra-low-power microcontrollers (Cortex M4 and M7) threshold. In particular, the large MCUNet [130] has the

best accuracy tradeoff and is right below the Cortex M7 memory threshold. Both versions of SqueezeNet [94] and MNasNet [194] have low accuracy, so they are unsuitable for practical application.

Visual Wake Word (VWW). We notice that no models are below the Cortex M0+ memory threshold, but the size of the RaScaNet models [222] shows that it would be reachable with further research. In the ultra-low-power range, MSNet [34] clearly provides the optimal size-performance tradeoff, but one could deploy the large RaScaNet for even lower power and acceptable accuracy. In comparison, the performance of MNasNet is less favorable. We also see that MobileNetV1 [14, 90] and MicroNet [14] display the worst size-performance tradeoff.

Google Speech Commands v2-12. In the extremely low-power range, we note that FastGRNN [111] offers the best tradeoff, while in the ultra-low-power range, μNAS displays once again the best tradeoff. ConvGRU 4-bits [115], ShallowRNN [46], Hello Edge DS-CNN [227], TinySpeech-Z [214], LSTM-KP [197], LMU-4 [24], FastRNN [111], DS-CNN [13] and all have acceptable performance, but are still less favorable than μNAS. In contrast, all versions of MicroNet present the least optimal performance once more, where the large MicroNet is even above the Cortex M4 threshold.

Preprocessing considerations. Preprocessing plays a critical role in TinyML performance evaluation, especially under strict memory and computation constraints. For the datasets reviewed here, typical preprocessing steps include resizing images (e.g., ImageNet and VWW resized to 96×96 or smaller resolutions), grayscale normalization (e.g., for MNIST), and basic feature normalization or feature extraction for speech datasets like Google Speech Commands v2-12. These preprocessing steps standardize input dimensions and dynamic ranges across models, ensuring fair comparisons. Consistent preprocessing is essential for reproducibility, allowing researchers and practitioners to benchmark models reliably and replicate performance results under comparable conditions, particularly when deployed on ultra-low-power devices with limited preprocessing capabilities.

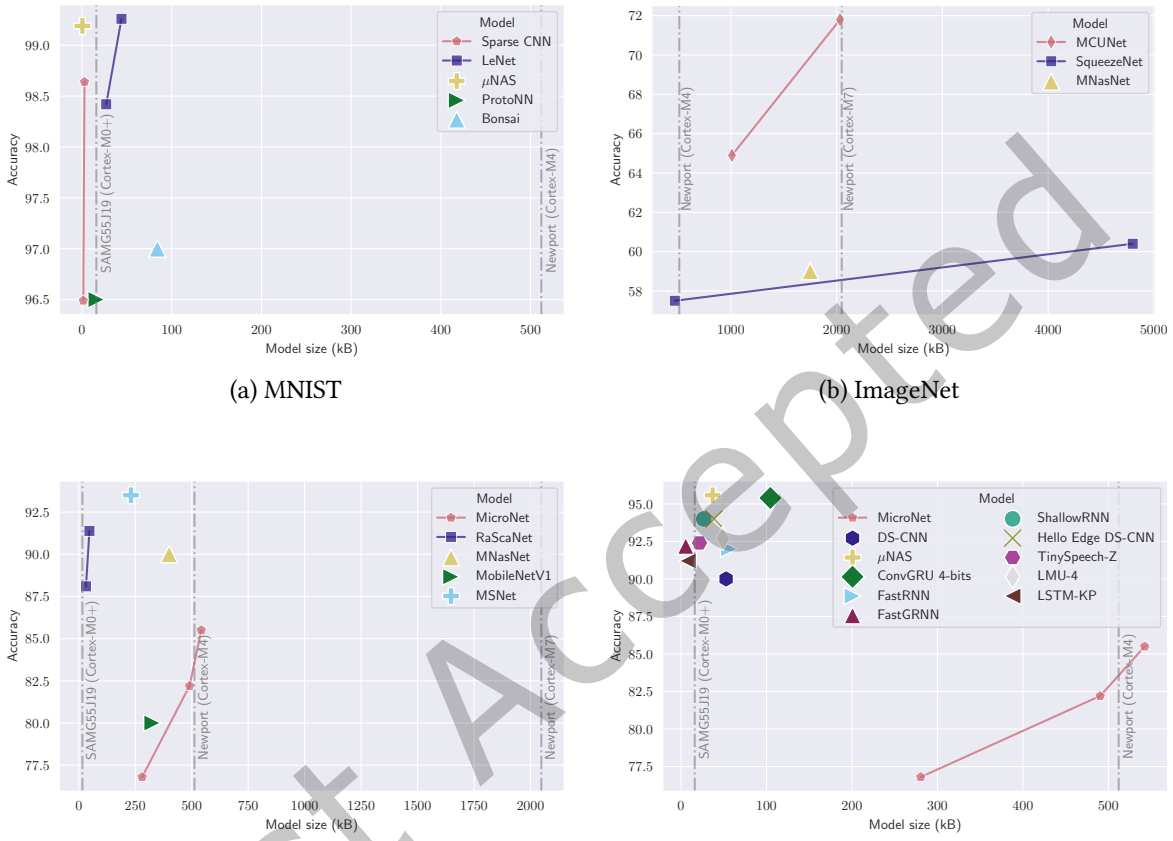
Beyond accuracy: additional metrics for TinyML. While accuracy remains the most commonly reported metric for evaluating TinyML models, several additional metrics are critical for assessing real-world feasibility in resource-constrained environments. *Latency* measures the time required for inference, which directly impacts the responsiveness of an embedded system. *Power consumption* quantifies the energy required for inference, affecting battery life and thermal characteristics. *Inference efficiency*, often expressed as operations-per-second or energy-per-inference, provides a holistic view of resource utilization. Although these metrics are not consistently reported across the literature reviewed, they are essential for a complete evaluation. Future research would benefit from systematically reporting these measures to support fair and comprehensive benchmarking of TinyML solutions.

Interpretation. We summarize the key trade-offs between accuracy, model size, and hardware feasibility through a series of plots in Figure 9. These plots present accuracy versus flash model size for the above common TinyML benchmark datasets, overlaid with vertical lines representing memory limits for different microcontroller classes (Cortex-M0+, M4, and M7). Figure 9 conveys the performance and compactness of each as well as their feasibility with respect to typical hardware constraints, allowing practitioners to directly assess which models can be deployed on specific platforms. It also shows the interplay between accuracy, model footprint, and MCU capabilities, helping guide efficient TinyML design decisions.

Among the standard datasets, TinyML models are able to comply with extreme-low power constraints as low as 8 kB for a speech recognition task and a simple image classification dataset, with a given tradeoff on accuracy. In this regard, further research efforts are required for an image recognition task and a more complex image classification problem. Otherwise, Cortex M4 is sufficient to run most models for all tasks with the best accuracy.

The industrial cost of exceeding a hardware memory threshold is high. Emphasizing the successful deployment of models on the most constrained microcontrollers is crucial, given the substantial economic impact. Even

though microcontroller power classes (extreme low-power and ultra-low-power) have minor price differences (ranging from 1 to 3 dollars) and are inexpensive, the price significance magnifies when considering the billions of annual unit market sales, resulting itself in billions of yearly savings. Thus, designing efficient models is critical for the TinyML industry, and inherently comes with a price tradeoff.



(a) MNIST (b) ImageNet

(c) Visual Wake Word (d) Google Speech Commands v2-12

Fig. 9. Flash model size versus accuracy on the four considered datasets. Vertical grey dashed lines indicate hardware storage limits for Cortex-M0+, Cortex-M4 and Cortex-M7 (see Table 4).

7 Conclusion and discussion

Summary. In Section 2 we presented the state of neural networks and motivated our interest in them for our applications, then we provided an overview of MEMS-based applications, emphasized the opportunities and challenges of our extremely low-power constraints, reinforcing the need for more TinyML research efforts in Section 3. In Section 4 we presented the existing methods to design efficient neural networks on ultra-low-power MCUs, and in Section 5 provided an overview of existing tools to deploy neural networks to enable TinyML applications. Finally, we examined the current limitations in the field of TinyML in Section 6.

Open challenges and research directions. TinyML is faced with a number of open challenges, where concrete research directions can be pursued. Ensuring the *robustness of TinyML models against adversarial attacks* remains a significant challenge. Adversarial attacks can manipulate input data to mislead the model, posing security risks in critical applications. Research could explore adapting *adversarial training* techniques to constrained devices, such as lightweight adversarial example generation and robust regularization during training, balancing robustness with resource limitations. Inference-time defences, such as input sanitization or lightweight anomaly detectors tailored for MCU hardware, are promising directions.

TinyML devices often operate in *dynamic environments* with fluctuating resource availability. To address this, future research could develop *adaptive resource management frameworks* where models adjust their computational and memory footprints on-the-fly. Example approaches include dynamically adjusting quantization levels, using scalable neural architectures (e.g., early exit networks) that can trade accuracy for latency under tight power budgets, and developing TinyNAS (Neural Architecture Search for TinyML) that produces models optimized for heterogeneous or dynamically constrained devices. Another important challenge is the lack of *standardized benchmarking* for comparing performance across diverse TinyML platforms and deployments. Future work should aim to define common evaluation protocols and shared datasets reflecting realistic, application-specific constraints (e.g., latency, memory, energy).

Concrete future research directions. Beyond these aspects, advancing TinyML will require closer *hardware-algorithm co-design*, extending beyond ARM architectures to include RISC-V and other heterogeneous low-power accelerators. The open and modular nature of RISC-V offers opportunities to explore instruction-level optimization, specialized neural operators, and energy-aware scheduling directly at the hardware level. At the software layer, establishing *standardized preprocessing pipelines* for sensor-rich data (e.g., MEMS, Inertial Measurement Unit (IMU), and audio streams) is essential to ensure reproducible and comparable performance across deployments. Furthermore, the definition of *composite evaluation metrics*, combining accuracy, latency, memory footprint, and energy per inference, will be critical to evaluate trade-offs and guide hardware-aware optimization. Finally, future TinyML systems should integrate *on-device learning, adaptive quantization, and privacy-preserving mechanisms*, enabling continuous, secure, and efficient intelligence directly at the edge. These research directions collectively aim to bridge the gap between theoretical efficiency and real-world deployment, unlocking the full potential of TinyML across hardware platforms and application domains.

Emerging trends and technologies. Several emerging trends and technologies are likely to impact the future of TinyML and could serve as enablers for addressing these challenges.

Edge AI and edge computing continue to grow, and new research could focus on collaborative TinyML approaches where multiple heterogeneous devices at the edge cooperate to share computational loads efficiently, for example through distributed inference or federated TinyML learning frameworks.

Quantum computing may offer opportunities in the longer term to accelerate TinyML model optimization and compression processes during training phases in the cloud before deployment. Research could investigate hybrid quantum-classical workflows for compressing neural networks to a size suitable for ultra-constrained deployment.

Custom hardware accelerators optimized for TinyML workloads remain an active area. Designing co-optimized hardware/software stacks, including MCU-specific instruction set extensions or specialized dataflow accelerators for sparse and quantized operations, can dramatically improve inference performance and energy efficiency. A promising research direction is to co-design lightweight TinyML models together with these emerging hardware accelerators using algorithm-hardware co-design methodologies.

These concrete directions highlight the need for continued interdisciplinary research spanning algorithms, systems, and hardware design, to unlock the full potential of TinyML applications on resource-constrained platforms.

Acknowledgments

This work is partially supported by ANR-21-JSTM-0001 grant.

References

[1] Abadi, Martín and Agarwal, Ashish and Barham, Paul and Brevdo, Eugene and Chen, Zhifeng and Citro, Craig and Corrado, Greg S and Davis, Andy and Dean, Jeffrey and Devin, Matthieu and others. 2016. TensorFlow: Large-scale Machine Learning on Heterogeneous Systems. (2016). https://arxiv.org/abs/1603.04467

[2] Giovanni Agosta, Andrea Galimberti, and Davide Zoni. 2025. Deep Learning on RISC-V Platforms at the Edge: A Perspective on the Hardware and Software Support. *Comput. Surveys* 58, 5 (2025), 1–37.

[3] Norah Alnaim and Maysam F. Abbod. 2019. Mini Gesture Detection Using Neural Networks Algorithms. *International Journal of Machine Learning and Computing* 9, 6 (2019), 782–787. doi:10.18178/ijmlc.2019.9.6.873

[4] Ali Alqahtani, Xianghua Xie, and Mark W. Jones. 2021. Literature Review of Deep Network Compression. *Informatics* 8, 4 (2021).

[5] Jose M Alvarez and Mathieu Salzmann. 2016. Learning the Number of Neurons in Deep Networks. In *Advances in Neural Information Processing Systems*, Vol. 29. Curran Associates, Inc.

[6] Jose M Alvarez and Mathieu Salzmann. 2017. Compression-Aware Training of Deep Networks. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

[7] Julyan Arbel, Konstantinos Pitas, Mariia Vladimirova, and Vincent Fortuin. 2024. A primer on Bayesian neural networks: review and debates. *Statist. Sci.* (2024).

[8] Sercan Ö Arık, Markus Kliegl, Rewon Child, Joel Hestness, Andrew Gibiansky, Ryan Prenger, and Adam Coates. 2017. Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting. *arXiv* abs/1703.05390 (2017).

[9] MohammadHossein AskariHemmat, Reyhane Askari Hemmat, Alex Hoffman, Ivan Lazarevich, Ehsan Saboori, Olivier Mastropietro, Sudhakar Sah, Yvon Savaria, and Jean-Pierre David. 2022. QReg: On Regularization Effects of Quantization. arXiv:2206.12372 [cs]

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[11] P. Baldi. 1995. Gradient Descent Learning Algorithm Overview: A General Dynamical Systems Perspective. *IEEE Transactions on Neural Networks* 6, 1 (1995), 182–195.

[12] Camille Ballas. 2022. *Inducing Sparsity in Deep Neural Networks through Unstructured Pruning for Lower Computational Footprint.* Ph. D. Dissertation. Dublin City University.

[13] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. MLPerf Tiny Benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks* (2021).

[14] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of machine learning and systems* 3 (2021), 517–532.

[15] 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).

[16] Luca Barbieri, Mattia Brambilla, Mario Stefanutti, Ciro Romano, Niccolò De Carlo, and Manuel Roveri. 2023. A tiny transformer-based anomaly detection framework for IoT solutions. *IEEE Open Journal of Signal Processing* 4 (2023), 462–478.

[17] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *National Academy of Sciences* 116, 32 (2019), 15849–15854.

[18] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. 2013. Estimating or Propagating Gradients through Stochastic Neurons for Conditional Computation. *arXiv* abs/1308.3432 (2013). arXiv:1308.3432

[19] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning Long-Term Dependencies with Gradient Descent Is Difficult. *IEEE Transactions on Neural Networks* 5, 2 (1994), 157–166.

[20] 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.

[21] Katyayani Bhardwaj, Aryan, and Ravindra Yadav. 2022. Single Input-Based CNN-LSTM and CNN-GRU Based HAR Using Wearable Sensors. In *Advancement in Electronics & Communication Engineering*.

[22] Christopher Bishop. 1995. Regularization and Complexity Control in Feed-Forward Networks. In *Proceedings International Conference on Artificial Neural Networks ICANN'95* (proceedings international conference on artificial neural networks icann'95 ed.), Vol. 1. EC2 et Cie, 141–148.

[23] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. 2018. Understanding Batch Normalization. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc.

[24] Peter Blouw, Gurshaant Malik, Benjamin Morcos, Aaron Voelker, and Chris Eliasmith. 2020. Hardware Aware Training for Efficient Keyword Spotting on General Purpose and Specialized Hardware. In *Research Symposium on Tiny Machine Learning*.

[25] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 1877–1901.

[26] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model Compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '06*. ACM Press, Philadelphia, PA, USA, 535.

[27] Rebekka Burkholz, Nilanjana Laha, Rajarshi Mukherjee, and Alkis Gotovos. 2022. On the Existence of Universal Lottery Tickets. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*.

[28] Roberto Cahuantzi, Xinye Chen, and Stefan Güttel. 2023. A comparison of LSTM and GRU networks for learning symbolic sequences. In *Science and Information Conference*. Springer, 771–785.

[29] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2020. ZeroQ: A Novel Zero Shot Quantization Framework. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 13166–13175.

[30] Carlos M Carvalho, Nicholas G Polson, and James G Scott. 2009. Handling sparsity via the horseshoe. In *Artificial intelligence and statistics*. PMLR, 73–80.

[31] Minmin Chen. 2018. MinimalRNN: Toward More Interpretable and Trainable Recurrent Neural Networks. *arXiv* abs/1711.06788 (2018).

[32] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, and Luis Ceze. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[33] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. 2024. A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 12 (2024), 10558–10578. doi:10.1109/TPAMI.2024.3447085

[34] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Harris Teague, Yiran Chen, and Hai Li. 2019. MSNet: Structural Wired Neural Architecture Search for Internet of Things. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. 2033–2036.

[35] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *arXiv* abs/1805.06085 (2018).

[36] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.

[37] Anna Choromańska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. 2014. The Loss Surfaces of Multilayer Networks. In *International Conference on Artificial Intelligence and Statistics*.

[38] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. 2019. Low-Bit Quantization of Neural Networks for Efficient Inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. 3009–3018.

[39] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual wake words dataset. *arXiv preprint arXiv:1906.05721* (2019).

[40] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

[41] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. In *Advances in Neural Information Processing Systems*, Vol. 28.

[42] G. Cybenko. 1989. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems* 2, 4 (1989), 303–314.

[43] Sajad Darabi, Mouloud Belbahri, Matthieu Courbariaux, and Vahid Partovi Nia. 2018. Regularized Binary Network Training. *arXiv* abs/1812.11800 (2018).

[44] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.

[45] Misha Denil, Babak Shakibi, Laurent Dinh, MarcAurelio Ranzato, and Nando de Freitas. 2013. Predicting Parameters in Deep Learning. In *Advances in Neural Information Processing Systems*, Vol. 26. Curran Associates, Inc.

[46] Don Dennis, Durmus Alp Emre Acar, Vikram Mandikal, Vinu Sankar Sadasivan, Venkatesh Saligrama, Harsha Vardhan Simhadri, and Prateek Jain. 2019. Shallow RNN: Accurate Time-series Classification on Resource Constrained Devices. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[47] Jeffrey L. Elman. 1990. Finding Structure in Time. *Cognitive Science* 14, 2 (1990), 179–211.

[48] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search. In *Automated Machine Learning: Methods, Systems, Challenges*. Springer International Publishing, Cham, 63–77.
[49] Jun Fang, Ali Shafiee, Hamzah Abdel-Aziz, David Thorsley, Georgios Georgiadis, and Joseph H. Hassoun. 2020. Post-Training Piecewise Linear Quantization for Deep Neural Networks. In *Computer Vision – ECCV 2020*. Vol. 12347. Springer International Publishing, Cham, 69–86.
[50] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. 2019. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems* 32 (2019).
[51] Igor Fedorov, Marko Stamenovic, Carl Jensen, Li-Chia Yang, Ari Mandell, Yiming Gan, Matthew Mattina, and Paul N. Whatmough. 2020. TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids. In *Interspeech 2020*. ISCA, 4054–4058.
[52] Jonas Fischer and Rebekka Burkholz. 2022. Plant 'n' Seek: Can You Find the Winning Ticket?. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*.
[53] Jonathan Frankle and Michael Carbin. 2018. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*.
[54] Pedro J. Freire, Antonio Napoli, Bernhard Spinnler, Michael Anderson, Diego Argüello Ron, Wolfgang Schairer, Thomas Bex, Nelson Costa, Sergei K. Turitsyn, and Jaroslaw E. Prilepsky. 2023. Reducing Computational Complexity of Neural Networks in Optical Channel Equalization: From Concepts to Implementation. *Journal of Lightwave Technology* 41, 14 (2023), 4557–4581. doi:10.1109/JLT.2023.3234327
[55] Fangcheng Fu, Yuzheng Hu, Yihan He, Jiawei Jiang, Yingxia Shao, Ce Zhang, and Bin Cui. 2020. Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript. In *International Conference on Machine Learning*. PMLR, 3304–3314.
[56] Kunihiko Fukushima. 1975. Cognitron: A Self-Organizing Multilayered Neural Network. *Biological Cybernetics* 20, 3 (1975), 121–136.
[57] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, New York, NY, USA, 1050–1059.
[58] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. *arXiv* abs/1902.09574 [cs, stat] (2019).
[59] Yimin Gao. 2022. *LiteQAIRISC: System-level Emulation of RISC-V Processor with AI and Mixed-precision Quantization Extensions*. Ph. D. Dissertation. University of Virginia.
[60] Thomas Garbay, Khalil Hachicha, Petr Dobias, Wilfried Dron, Pedro Lusich, Imane Khalis, Andrea Pinna, and Bertrand Granado. 2022. Accurate Estimation of the CNN Inference Cost for TinyML Devices. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 1–6.
[61] Angelo Garofalo and Luca Benini. 2025. Leveraging RISC-V for HW/SW Codesign of Flexible and Efficient TinyML SoCs. *IEEE Design & Test* 42, 5 (2025), 8–26. doi:10.1109/MDAT.2025.3573686
[62] F.A. Gers, J. Schmidhuber, and F. Cummins. 1999. Learning to Forget: Continual Prediction with LSTM. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, Vol. 2. 850–855 vol.2.
[63] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. 2003. Learning Precise Timing with Lstm Recurrent Networks. *Journal of Machine Learning Research* 3 (2003), 115–143.
[64] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2022. A Survey of Quantization Methods for Efficient Neural Network Inference. In *Low-Power Computer Vision*. Chapman and Hall/CRC, 291–326.
[65] Soumya Ghosh, Jiayu Yao, and Finale Doshi-Velez. 2019. Model Selection in Bayesian Neural Networks via Horseshoe Priors. *Journal of Machine Learning Research* 20, 182 (2019), 1–46.
[66] Aditya Sharad Golatkar, Alessandro Achille, and Stefano Soatto. 2019. Time Matters in Regularizing Deep Networks: Weight Decay and Data Augmentation Affect Early Learning Dynamics, Matter Little Near Convergence. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.
[67] Anna Golubeva, Guy Gur-Ari, and Behnam Neyshabur. 2021. Are Wider Nets Better given the Same Number of Parameters?. In *International Conference on Learning Representations*.
[68] Yuan Gong and Christian Poellabauer. 2018. Impact of Aliasing on Deep CNN-Based End-to-End Acoustic Models. In *Proc. Interspeech 2018*. 2698–2702.
[69] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA, USA.
[70] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.
[71] Alex Graves. 2011. Practical Variational Inference for Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 24. Curran Associates, Inc.
[72] Alex Graves and Navdeep Jaitly. 2014. Towards End-to-End Speech Recognition with Recurrent Neural Networks. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*. PMLR, Bejing, China, 1764–1772.
[73] R.M. Gray and D.L. Neuhoff. 1998. Quantization. *IEEE Transactions on Information Theory* 44, 6 (1998), 2325–2383.
[74] Yunhui Guo. 2018. A Survey on Methods and Theories of Quantized Neural Networks. *arXiv* abs/1808.04752 (2018).

[75] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1331–1340.
[76] M. Hagiwara. 1993. Removal of Hidden Units and Weights for Back Propagation Networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, Vol. 1. 351–354 vol.1.
[77] Hui Han and Julien Siebert. 2022. TinyML: A Systematic Review and Synthesis of Existing Research. In *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. 269–274.
[78] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.
[79] Qusay F. Hassan and Assim Sagahyroon. 2025. RISC-V: A Comprehensive Overview of an Emerging ISA for the AI-IoT Era. *Advances in the Internet of Things* (2025). https://www.taylorfrancis.com/chapters/edit/10.1201/9781003506638-15/risc-qusay-hassan-assim-sagahyroon
[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs]
[81] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[82] Yang He and Lingao Xiao. 2024. Structured Pruning for Deep Convolutional Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 5 (2024), 2900–2919. doi:10.1109/TPAMI.2023.3334614
[83] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 1398–1406.
[84] Joel C. Heck and Fathi M. Salem. 2017. Simplified Minimal Gated Unit Variations for Recurrent Neural Networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, Boston, MA, USA, 1593–1596.
[85] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. *arXiv* abs/1503.02531 [cs, stat] (2015).
[86] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.
[87] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *The Journal of Machine Learning Research* 22, 1 (2021), 10882–11005.
[88] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer Feedforward Networks Are Universal Approximators. *Neural Networks* 2, 5 (1989), 359–366.
[89] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. arXiv:1905.02244 [cs]
[90] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs]
[91] Yanbo Huang. 2009. Advances in Artificial Neural Networks – Methodological Development and Application. *Algorithms* 2, 3 (2009), 973–1007.
[92] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 4114–4122.
[93] David A. Huffman. 2006. A Method for the Construction of Minimum-Redundancy Codes. *Resonance* 11, 2 (2006), 91–99.
[94] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5MB Model Size. *arXiv* abs/1602.07360 (2016).
[95] IEEE. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84.
[96] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, Lille, France, 448–456.
[97] 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*. 2704–2713.
[98] Vijay Janapa Reddi, Alexander Elium, Shawn Hymel, David Tischler, Daniel Situnayake, Carl Ward, Louis Moreau, Jenny Plunkett, Matthew Kelcey, Mathijs Baaijens, et al. 2023. Edge Impulse: An MLOps Platform for Tiny Machine Learning. *Proceedings of Machine Learning and Systems* (2023).
[99] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 4163–4174. doi:10.18653/v1/2020.findings-emnlp.372

[100] Victor JB Jung, Alessio Burrello, Moritz Scherer, Francesco Conti, and Luca Benini. 2024. Optimizing the deployment of tiny transformers on low-power MCUs. *IEEE Trans. Comput.* (2024).
[101] Eiman Kanjo. 2022. Sensing on the Edge: Smartening up Sensors. In *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*. 1–1.
[102] Kenji Kawaguchi and Jiaoyang Huang. 2019. Gradient Descent Finds Global Minima for Generalizable Deep Neural Networks of Practical Sizes. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 92–99.
[103] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. 2020. A Survey of the Recent Architectures of Deep Convolutional Neural Networks. *Artificial Intelligence Review* 53, 8 (2020), 5455–5516.
[104] Mohammad Emtiyaz Khan and Håvard Rue. 2023. The Bayesian Learning Rule. *Journal of Machine Learning Research* 1, 4 (2023), 5.
[105] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. 2019. ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*. IEEE, 313–317.
[106] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. 2022. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access* (2022), 13.
[107] Raghuraman Krishnamoorthi. 2018. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper. *arXiv* abs/1806.08342 [cs, stat] (2018).
[108] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. (2009).
[109] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 25. Curran Associates, Inc.
[110] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1935–1944.
[111] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. 2018. FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 9031–9042.
[112] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv* abs/1801.06601 [cs] (2018).
[113] Gerhard Lammel. 2015. The Future of MEMS Sensors in Our Connected World. In *2015 28th IEEE International Conference on Micro Electro Mechanical Systems (MEMS)*. 61–64.
[114] Minh Tri Lê and Julyan Arbel. 2023. TinyMLOps for real-time ultra-low power MCUs applied to frame-based event classification. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 148–153.
[115] Minh Tri Lê, Etienne de Foras, and Julyan Arbel. 2023. Regularization for hybrid n-bit weight quantization of neural networks on ultra-low power microcontrollers. In *International Conference on Artificial Neural Networks*. Springer, 435–446.
[116] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. 2015. Speeding-up Convolutional Neural Networks Using Fine-Tuned CP-Decomposition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.
[117] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.
[118] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[119] Jun Haeng Lee, Sangwon Ha, Saerom Choi, Won-Jo Lee, and Seungwon Lee. 2018. Quantization for Rapid Deployment of Deep Neural Networks. *arXiv* abs/1810.05488 (2018). arXiv:1810.05488
[120] Sam Leroux, Pieter Simoens, Meelis Lootus, Kartik Thakore, and Akshay Sharma. 2022. TinyMLOps: Operational challenges for widespread edge AI Adoption. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1003–1010.
[121] Guoqing Li, Rengang Li, Tuo Li, Tinghuan Chen, Meng Zhang, and Henk Corporaal. 2025. Algorithm-Hardware Co-design for Accelerating Depthwise Separable CNNs. *ACM Transactions on Design Automation of Electronic Systems* 30, 2 (2025), 1–22.
[122] Yuanzhi Li and Yingyu Liang. 2018. Learning Overparameterized Neural Networks via Stochastic Gradient Descent on Structured Data. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 8168–8177.
[123] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. 2020. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning*. PMLR, 5958–5968.
[124] 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. doi:10.1016/j.neucom.2021.07.045

[125] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. 2021. μNAS: Constrained Neural Architecture Search for Microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems (EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 70–79.
[126] Edgar Liberis and Nicholas D. Lane. 2020. Neural Networks on Microcontrollers: Saving Memory at Inference via Operator Reordering. *arXiv* abs/1910.05110 (2020).
[127] Edgar Liberis and Nicholas D Lane. 2023. Differentiable neural network pruning to enable smart applications on microcontrollers. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 4 (2023), 1–19.
[128] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, New York, NY, USA, 2849–2858.
[129] Hongzhou Lin and Stefanie Jegelka. 2018. ResNet with One-Neuron Hidden Layers Is a Universal Approximator. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 6172–6181.
[130] Ji Lin, Wei-Ming Chen, Yujun Lin, john cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 11711–11722.
[131] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2022. A Survey of Transformers. *AI Open* 3 (2022), 111–132.
[132] Yijia Liu, Wanxiang Che, Bing Qin, and Ting Liu. 2020. Exploring Segment Representations for Neural Semi-Markov Conditional Random Fields. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 28 (2020), 813–824.
[133] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. 2017. Learning Efficient Convolutional Networks through Network Slimming. In *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE Computer Society, Los Alamitos, CA, USA, 2755–2763.
[134] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2018. Rethinking the Value of Network Pruning. In *International Conference on Learning Representations*.
[135] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. 2018. Bi-Real Net: Enhancing the Performance of 1-Bit CNNs with Improved Representational Capability and Advanced Training Algorithm. In *Computer Vision – ECCV 2018*. Springer International Publishing, Cham, 747–763.
[136] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. 2018. Relaxed Quantization for Discretized Neural Networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.
[137] Christos Louizos, Karen Ullrich, and Max Welling. 2017. Bayesian compression for deep learning. *Advances in neural information processing systems* 30 (2017).
[138] Christos Louizos, Max Welling, and Diederik P. Kingma. 2018. Learning Sparse Neural Networks through L_0 Regularization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.
[139] Limeng Lu, Chuanlin Zhang, Kai Cao, Tao Deng, and Qianqian Yang. 2022. A Multichannel CNN-GRU Model for Human Activity Recognition. *IEEE access : practical innovations, open solutions* 10 (2022), 66797–66810.
[140] JY Rumbo-Morales M Beltrán-Escobar, TE Alarcón. 2024. A Review on Resource-Constrained Embedded Vision Systems-based TinyML for Robotic Applications. *Algorithms* 17, 11 (2024). https://www.mdpi.com/1999-4893/17/11/476
[141] Jianjia Ma. 2020. A Higher-Level Neural Network Library on Microcontrollers (NNoM). Zenodo.
[142] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning*, Vol. 30. Atlanta, GA, 3.
[143] Alexis Maras. 2024. *Extending RISC-V ISA for Fine-Grained Mixed-Precision in Neural Networks*. Ph. D. Dissertation. National Technical University of Athens. https://dspace.lib.ntua.gr/xmlui/handle/123456789/59804
[144] Warren S. McCulloch and Walter Pitts. 1943. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.
[145] Daniel Menard, Daniel Chillet, and Olivier Sentieys. 2006. Floating-to-Fixed-Point Conversion for Digital Signal Processors. *EURASIP Journal on Advances in Signal Processing* 2006, 1 (2006), 096421.
[146] Xiangming Meng, Roman Bachmann, and Mohammad Emtiyaz Khan. 2020. Training binary neural networks using the bayesian learning rule. In *International conference on machine learning*. PMLR, 6852–6861.
[147] Gaurav Menghani. 2023. Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better. *Acm Computing Surveys* 55, 12, Article 259 (2023).
[148] Toby J Mitchell and John J Beauchamp. 1988. Bayesian variable selection in linear regression. *Journal of the american statistical association* 83, 404 (1988), 1023–1032.
[149] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. 2016. Convolutional Neural Networks Using Logarithmic Data Representation. *arXiv* abs/1603.01025 (2016).
[150] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Omnipress, Madison, WI, USA, 807–814.

[151] Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. 2021. Deep double descent: Where bigger models and more data hurt. *Journal of Statistical Mechanics: Theory and Experiment* 2021, 12 (2021), 124003.
[152] James O' Neill. 2020. An Overview of Neural Network Compression. *arXiv* abs/2006.03669 [cs, stat] (2020).
[153] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry P Vetrov. 2017. Structured bayesian pruning via log-normal multiplicative noise. *Advances in Neural Information Processing Systems* 30 (2017).
[154] Pierre-Emmanuel Novac, Ghouthi Boukli Hacene, Alain Pegatoquet, Benoît Miramond, and Vincent Gripon. 2021. Quantization and Deployment of Deep Neural Networks on Microcontrollers. *Sensors* 21, 9 (2021), 2984.
[155] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. 2015. Tensorizing neural networks. *Advances in neural information processing systems* 28 (2015).
[156] Steven J. Nowlan and Geoffrey E. Hinton. 1992. Simplifying Neural Networks by Soft Weight-Sharing. *Neural Computation* 4, 4 (1992), 473–493.
[157] OpenAI. 2023. GPT-4 Technical Report. *arXiv* abs/2303.08774 (2023).
[158] Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. 2023. Cheshire: A Lightweight, Linux-capable RISC-V Host Platform for Domain-Specific Accelerator Plug-in. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3777–3781.
[159] Theodore Papamarkou, Maria Skoularidou, Konstantina Palla, Laurence Aitchison, Julyan Arbel, David Dunson, Maurizio Filippone, Vincent Fortuin, Philipp Hennig, Aliaksandr Hubin, Alexander Immer, Theofanis Karaletsos, Mohammad Emtiyaz Khan, Agustinus Kristiadi, Yingzhen Li, Jose Miguel Hernandez Lobato, Stephan Mandt, Christopher Nemeth, Michael A. Osborne, Tim G. J. Rudner, David Rügamer, Yee Whye Teh, Max Welling, Andrew Gordon Wilson, and Ruqi Zhang. 2024. Position Paper: Bayesian Deep Learning in the Age of Large-Scale AI. arXiv:2402.00809 [cs.LG]
[160] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035.
[161] Tomaso Poggio, Andrzej Banburski, and Qianli Liao. 2020. Theoretical Issues in Deep Networks. *Proceedings of the National Academy of Sciences* 117, 48 (2020), 30039–30045.
[162] Tomaso Poggio, Qianli Liao, and Andrzej Banburski. 2020. Complexity Control by Gradient Descent in Deep Networks. *Nature Communications* 11, 1 (2020), 1027.
[163] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. In *International Conference on Learning Representations*.
[164] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary Neural Networks: A Survey. *Pattern Recognition* 105 (2020), 107281.
[165] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. 2020. What's Hidden in a Randomly Weighted Neural Network?. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11890–11899.
[166] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Computer Vision – ECCV 2016*. Springer International Publishing, Cham, 525–542.
[167] Partha Pratim Ray. 2022. A Review on TinyML: State-of-the-art and Prospects. *Journal of King Saud University - Computer and Information Sciences* 34, 4 (2022), 1595–1623.
[168] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. MLperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.
[169] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. 2023. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Transactions on Intelligent Systems and Technology* 14, 6 (2023), 1–50.
[170] Frank Rosenblatt. 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological review* 65, 6 (1958), 386.
[171] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning Representations by Back-Propagating Errors. *Nature* 323, 6088 (1986), 533–536.
[172] Avyay Sah, Soham Chatterjee, and Archana Vaidheeswaran. 2022. TinyMLOps: Overview, Challenges and Implementation. (2022). TinyML Summit.
[173] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. 2022. Machine Learning for Microcontroller-Class Hardware: A Review. *IEEE Sensors Journal* 22, 22 (2022), 21362–21390.
[174] Tara N. Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. 2013. Low-Rank Matrix Factorization for Deep Neural Network Training with High-Dimensional Output Targets. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 6655–6659.
[175] S. Sakib, N. Ahmed, A. J. Kabir, and H. Ahmed. 2018. An Overview of Convolutional Neural Network: Its Architecture and Applications. *Preprints.org* 2018, 2018110546 (2018).

[176] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.
[177] Martin Scherer. 2024. *Hardware-Software Co-Design for Energy-Efficient Neural Network Inference at the Extreme Edge*. Ph. D. Dissertation. ETH Zurich. https://www.research-collection.ethz.ch/handle/20.500.11850/698281
[178] Moritz Scherer, Cristian Cioflan, Michele Magno, and Luca Benini. 2024. Work In Progress: Linear Transformers for TinyML. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–2.
[179] Nikolaos Schizas, Aristeidis Karras, Christos Karras, and Spyros Sioutas. 2022. TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review. *Future Internet* 14, 12 (2022).
[180] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc.
[181] Iman Sharifirad, Jalil Boudjadar, and Peter Gorm Larsen. 2025. TinyML for Computation-aware Transformer-based Anomaly Detection in Internal Combustion Systems. In *2025 IEEE 23rd World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE, 000141–000146.
[182] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A Survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* 6, 1 (2019), 60.
[183] Karen Simonyan and Andrew Zisserman. 2014. Two-Stream Convolutional Networks for Action Recognition in Videos. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 568–576.
[184] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.
[185] Tuomo Sipola, Janne Alatalo, Tero Kokkonen, and Mika Rantonen. 2022. Artificial Intelligence in the IoT Era: A Review of Edge AI Hardware and Software. In *2022 31st Conference of Open Innovations Association (FRUCT)*. 320–331.
[186] J. Sjöberg and L. Ljung. 1992. Overtraining, Regularization, and Searching for Minimum in Neural Networks. *IFAC Proceedings Volumes* 25, 14 (1992), 73–78.
[187] Max Sponner, Bernd Waschneck, and Akash Kumar. 2020. Compiler Toolchains for Deep Learning Workloads on Embedded Platforms. In *Research Symposium on Tiny Machine Learning*.
[188] Suraj Srinivas and R. Venkatesh Babu. 2015. Data-Free Parameter Pruning for Deep Neural Networks. In *British Machine Vision Conference (BMVC)*.
[189] Suraj Srinivas, Akshayvarun Subramanya, and R Venkatesh Babu. 2017. Training sparse neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 138–145.
[190] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.
[191] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
[192] Thanaphon Suwannaphong, Ferdian Jovan, Ian Craddock, and Ryan McConville. 2025. Optimising TinyML with quantization and distillation of transformer and mamba models for indoor localisation on edge devices. *Scientific Reports* 15, 1 (2025), 10081.
[193] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper With Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
[194] 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 (CVPR)*.
[195] Mingxing Tan and Quoc V Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.
[196] Urmish Thakker, Jesse Beu, Dibakar Gope, Ganesh Dasika, and Matthew Mattina. 2020. Rank and run-time aware compression of NLP Applications. In *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*. Association for Computational Linguistics, 8–18.
[197] Urmish Thakker, Igor Fedorov, Chu Zhou, Dibakar Gope, Matthew Mattina, Ganesh Dasika, and Jesse Beu. 2021. Compressing RNNs to Kilobyte Budget for IoT Devices Using Kronecker Products. *J. Emerg. Technol. Comput. Syst.* 17, 4, Article 46 (jul 2021), 18 pages.
[198] Theano Development Team. 2016. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv* abs/1605.02688 (2016).
[199] Lukas Timpl, Rahim Entezari, Hanie Sedghi, Behnam Neyshabur, and Olga Saukh. 2022. Understanding the Effect of Sparsity on Neural Networks Robustness. arXiv:2206.10915 [cs.CV]

[200] Karen Ullrich, Edward Meeds, and Max Welling. 2016. Soft Weight-Sharing for Neural Network Compression. In *International Conference on Learning Representations*.
[201] Hasan Unlu. 2020. Efficient Neural Network Deployment for Microcontroller. *arXiv* abs/2007.01348 [cs] (2020).
[202] Sunil Vadera and Salem Ameen. 2022. Methods for Pruning Deep Neural Networks. *IEEE Access* 10 (2022), 63280–63300. doi:10.1109/ACCESS.2022.3182659
[203] Mart Van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, and Max Welling. 2020. Bayesian bits: Unifying quantization and pruning. *Advances in neural information processing systems* 33 (2020), 5741–5752.
[204] Vladimir Vapnik. 2013. *The nature of statistical learning theory*. Springer Science & Business Media.
[205] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).
[206] Vishesh Verma, Thomas Tracy II, and Michael R. Stan. 2022. Extrem-Edge—Extensions to RISC-V for Energy-Efficient ML Inference at the Edge of IoT. *Sustainable Computing: Informatics and Systems* 35 (2022), 100743. doi:10.1016/j.suscom.2022.100743
[207] Mathukumalli Vidyasagar. 2013. *Learning and generalisation: with applications to neural networks*. Springer Science & Business Media.
[208] Mariia Vladimirova, Julyan Arbel, and Stéphane Girard. 2021. Bayesian Neural Network Unit Priors and Generalized Weibull-tail Property. In *Asian Conference on Machine Learning, ACML 2021, 17-19 November 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 157)*. PMLR, 1397–1412.
[209] Mariia Vladimirova, Jakob Verbeek, Pablo Mesejo, and Julyan Arbel. 2019. Understanding Priors in Bayesian Neural Networks at the Unit Level. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 6458–6467.
[210] Christopher A. Walsh. 2013. Peter Huttenlocher (1931–2013). *Nature* 502, 7470 (2013), 172–172.
[211] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).
[212] P. Warden and D. Situnayake. 2020. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly.
[213] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 2082–2090.
[214] Alexander Wong, Mahmoud Famouri, Maya Pavlova, and Siddharth Surana. 2020. Tinyspeech: Attention condensers for deep speech recognition neural networks on edge devices. *arXiv* abs/2008.04245 (2020).
[215] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *arXiv* abs/2004.09602 (2020).
[216] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. 2018. Deep k-Means: Re-Training and Parameter Sharing with Harder Cluster Assignments for Compressing Deep Convolutions. In *International Conference on Machine Learning*. PMLR, 5363–5372.
[217] Jian Xue, Jinyu Li, and Yifan Gong. 2013. Restructuring of Deep Neural Network Acoustic Models with Singular Value Decomposition. In *Interspeech*.
[218] Jianlei Yang, Jiacheng Liao, Fanding Lei, Meichen Liu, Junyi Chen, Lingkun Long, Han Wan, Bei Yu, and Weisheng Zhao. 2023. TinyFormer: Efficient Transformer Design and Deployment on Tiny Devices. *arXiv preprint arXiv:2311.01759* (2023).
[219] Yibo Yang, Robert Bamler, and Stephan Mandt. 2020. Variational bayesian quantization. In *International Conference on Machine Learning*. PMLR, 10670–10680.
[220] Penghang Yin, Shuai Zhang, Jiancheng Lyu, Stanley J. Osher, Yingyong Qi, and Jack Xin. 2018. BinaryRelax: A Relaxation Approach for Training Deep Neural Networks with Quantized Weights. *SIAM J. Imaging Sci.* 11, 4 (2018), 2205–2223.
[221] Joseph Yiu. 2019. Cortex-M Resources - Processor Documentation.
[222] Jaehyoung Yoo, Dongwook Lee, Changyong Son, Sangil Jung, ByungIn Yoo, Changkyu Choi, Jae-Joon Han, and Bohyung Han. 2021. RaScaNet: Learning Tiny Models by Raster-Scanning Images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 13673–13682.
[223] K Schleiser Z Huang, K Zandberg. 2025. RIOT-ML: Toolkit for Over-the-Air Secure Updates and Performance Evaluation of TinyML Models. *Annals of Telecommunications* (2025). doi:10.1007/s12243-024-01041-5
[224] Hadi Al Zein, Mohamad Aoude, and Youssef Harkous. 2022. Implementation and Optimization of Neural Networks for Tiny Hardware Devices. In *2022 International Conference on Smart Systems and Power Management (IC2SPM)*. 191–196. doi:10.1109/IC2SPM56638.2022.9988992
[225] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. Understanding Deep Learning (Still) Requires Rethinking Generalization. *Communications of The Acm* 64, 3 (2021), 107–115.
[226] Xinyu Zhang, Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das. 2021. Training Deep Neural Networks with Joint Quantization and Pruning of Weights and Activations. *arXiv* abs/2110.08271 (2021). arXiv:2110.08271

[227] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2018. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv* abs/1710.01878 [cs, eess] (2018). arXiv:1711.07128 [cs, eess]
[228] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.
[229] Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. 2016. Minimal Gated Unit for Recurrent Neural Networks. *International Journal of Automation and Computing* 13, 3 (2016), 226–234.
[230] Jianxiong Zhu, Xinmiao Liu, Qiongfeng Shi, Tianyiyi He, Zhongda Sun, Xinge Guo, Weixin Liu, Othman Bin Sulaiman, Bowei Dong, and Chengkuo Lee. 2020. Development Trends and Perspectives of Future Sensors and MEMS/NEMS. *Micromachines* 11, 1 (2020).
[231] Michael Zhu and Suyog Gupta. 2017. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. *arXiv* abs/1710.01878 [cs, stat] (2017). arXiv:1710.01878 [cs, stat]
[232] Davide Zoni and Andrea Galimberti. 2022. Cost-effective fixed-point hardware support for RISC-V embedded systems. *Journal of Systems Architecture* 126 (2022), 102476.

Received 17 October 2024; revised 15 January 2026; accepted 26 January 2026