IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 October 2024, accepted 28 November 2024, date of publication 9 December 2024, date of current version 17 December 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3512860

TOPICAL REVIEW

A Holistic Review of the TinyML Stack for Predictive Maintenance

EMIL NJOR, MOHAMMAD AMIN HASANPOUR, (Graduate Student Member, IEEE), JAN MADSEN, (Senior Member, IEEE), AND XENOFON FAFOUTIS, (Senior Member, IEEE)
Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2800 Kongens Lyngby, Denmark

Corresponding author: Emil Njor (emjn@dtu.dk)

This work was supported by the Innovation Fund Denmark for the Project Digital Research Centre Denmark (DIREC) under Grant 9142-00001B.

ABSTRACT Downtime caused by failing equipment can be extremely costly for organizations. Predictive Maintenance (PdM), which uses data to predict when maintenance should be conducted, is an essential tool for increasing safety, maximizing uptime and minimizing costs. Contempoary PdM systems primarily have sensors collect information about the equipment under observation. This information is afterwards transmitted off the device for processing at a high-performance computer system. While this can allow high-quality predictions, it also imposes barriers that keep some organisations from adopting PdM. For example, some applications prevent data transmission off sensor devices due to regulatory or infrastructure limitations. Being able to process the collected information right at the sensor device is, therefore, desirable in many sectors - something that recent progress in the field of TinyML promises to deliver. This paper investigates the intersection between PdM and TinyML and explores how TinyML can enable many new PdM applications. We consider a holistic view of TinyML-based PdM, focusing on the full stack of Machine Learning (ML) models, hardware, toolchains, data and PdM applications. Our main findings are that each part of the TinyML stack has received varying degrees of attention. In particular, ML models and their optimisations have seen a lot of attention, while data optimisations and TinyML datasets lack contributions. Furthermore, most TinyML research focuses on image and audio classification, with little attention paid to other application areas such as PdM. Based on our observations, we suggest promising avenues of future research to scale and improve the application of TinyML to PdM.

INDEX TERMS Embedded AI, embedded machine learning, optimization, predictive maintenance, resource-constrained systems, TinyML.

I. INTRODUCTION

Equipment failure can cause extremely costly downtime for organizations. In 2021, Amazon lost approximately $2 million from lost sales because of a 13-minute downtime [1], and a study by the Ponemon Institute found that organizations, on average, lose $138.000 per hour of data centre downtime [2]. Unsurprisingly, organizations are willing to go to great lengths to reduce such downtime. According to a 2000 article, some organizations spend up to 70% of total production costs on maintenance [3]. Given such high potential costs, it is paramount that organizations employ a maintenance strategy that maximize equipment uptime while also keeping maintenance costs low.

According to Ran et al. [2], there are three main categories of maintenance strategies.

Reactive Maintenance: In reactive maintenance, equipment is replaced after a failure. While this avoids unnecessary maintenance, it can lead to costly downtime.

Preventive Maintenance: In preventive maintenance, equipment is maintained according to a predefined schedule. This strategy can be costly and wasteful as it may replace perfectly working equipment. Preventive Maintenance also provides no guarantee that failures do not occur before the scheduled maintenance.

Predictive Maintenance: In PdM, equipment is monitored using sensors to predict when maintenance is required.

The associate editor coordinating the review of this manuscript and approving it for publication was Mu-Yen Chen.

Based on these metrics, it is often possible to predict when the machine should be maintained to avoid failure and costly downtime while keeping maintenance costs low.

PdM clearly provides the most attractive maintenance strategy combining low failure rates and maintenance costs. Unfortunately, it also comes with large upfront costs and is hard to implement successfully due to its complexity. Because of this, many organizations still employ reactive or preventive maintenance today.

There are mainly three approaches to implementing PdM systems [2]:

Knowledge-based: This PdM approach establishes rules or physical models to predict failures. Establishing comprehensive rules or physical models for complex systems can be challenging. Therefore, this approach is mainly applied to simple systems.

Traditional Machine Learning (ML): This PdM approach uses traditional ML models to predict failures. For example, a decision tree or k-means. Traditional ML often rely on significant feature engineering to achieve high predictive performance. Even so, traditional ML approaches may not be suitable for complex problems such as the classification of images.

Deep Learning: This PdM approach uses deep Neural Network (NN) models to predict failures. Unlike traditional ML, deep learning works well with unstructured data and, thus, does not require significant feature engineering work. Deep learning often consumes more system resources than traditional ML but can solve complex problems where traditional ML struggles.

The output of PdM models can be of different forms [2]:

Classification: Outputs a classification of the PdM status of the input data. For example a binary prediction of whether there is an impending failure within an established time frame.

Anomaly Detection: Outputs whether the input data is normal or abnormal, considering previous data. If the behaviour is abnormal, a maintenance team can be sent to investigate the anomaly.

Regression: Outputs an estimate of the Remaining Useful Life (RUL). A maintenance team can be sent when the RUL approaches low levels.

A further distinction is possible in anomaly detection, where anomalies can be split into three subcategories [4]:

Point Anomalies: An anomalous input data sample considering previously seen samples.

Contextual Anomalies: An input data sample that is anomalous in its context. E.g. a high-temperature reading in industrial machinery during downtime hours.

Collective Anomalies: A collection of input data samples that are anomalous as a collective. E.g. a large amount of failed login attempts on a website.

Today, most PdM systems are deployed in the cloud or on powerful computers. However, the data that feeds into these PdM models is almost exclusively generated by small sensor devices. Therefore, data has to be sent over a network for processing. This introduces several drawbacks, e.g.:

Energy Consumption: While on-device computation is energy efficient, sending data over a network consumes a significant amount of energy. This can be a problem for devices relying on batteries or energy harvesting for operation [5].

Security and Privacy: Data sent over a network can be compromised. While several techniques exist to alleviate this problem, it is an inherent risk in network-based solutions. This makes these solutions hard or impossible to apply in highly regulated industries.

Latency: Network communication induces a non-zero and often unpredictable latency. This can be intolerable in some applications requiring real-time processing.

Reliability: A network-based system will be less reliable than a standalone system, as it relies on a working network connection and a responsive endpoint. This can be especially problematic for systems deployed in rural areas, at sea, or in space.

Based on these drawbacks, it is desirable, or even essential, for some industries to deploy PdM systems directly on sensor devices. These devices, unfortunately, lack the resources to naively deploy complicated physical models, traditional ML models and deep learning models.

The field of TinyML is dedicated to lowering the resource requirements of traditional ML and deep learning models and making them fit within resource-constrained devices [6]. This goal is pursued through different components of the TinyML stack, with the layers illustrated in Figure 1. TinyML can, therefore, help PdM overcome the above-mentioned drawbacks and enable groundbreaking applications. Naturally, making machine learning run under extreme resource constraints introduces some challenges.

Limited Compute: Few Computational Resources are available under the power limits enforced by TinyML. Inference of models, therefore, takes significantly longer than in the cloud or desktops.

Limited Memory: Low-power devices contain an extremely limited amount of memory, constraining the size of models deployable with TinyML.

No Operating System: Little or no resources in a TinyML system can be spared for an operating system. Therefore TinyML can not rely on standard operating system features such as dynamic memory allocation.

Market Heterogeneity: There is a wide variety of low-power devices on the market, and little to no interoperability between them. Therefore, significant engineering work is required to make well-working TinyML systems.

Additionally, in many industrial applications, most computation and memory resources are already allocated to dedicated tasks, leaving even less room for AI operations. A holistic view of the TinyML stack seen in Figure 1 is necessary to deliver well-performing systems under these

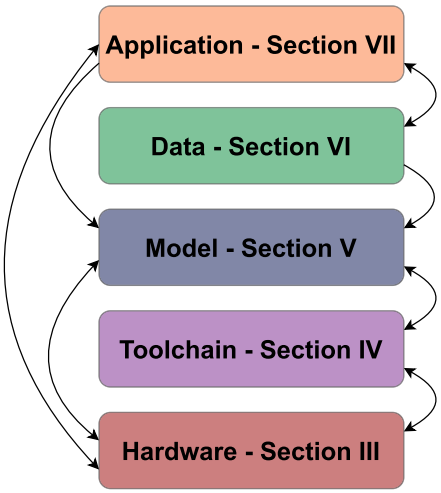

FIGURE 1. The TinyML Stack (and following sections of the paper). Arrows indicate that a part of the stack influences another part.

challenges. This review is dedicated to that task and considers optimisations to hardware, ML models, toolchains and the datasets that are required for successfully using TinyML for PdM. More specifically, our contribution is to provide a comprehensive perspective of the following aspects of the intersection of TinyML and PdM:

Hardware (Section III): We describe how various types of hardware strike different trade-offs between energy efficiency, flexibility, usability, and cost.

Toolchain (Section IV): We provide information about the available toolchains for developing and deploying TinyML models and their characteristics.

Model (Section V): We discuss various families of machine learning models and their suitability for TinyML-based PdM. Further, available TinyML optimisation techniques are studied.

Data (Section VI): We elaborate on the importance of data in TinyML systems and how it can be optimised for PdM applications.

Application (Section VII): We introduce various objectives of PdM systems alongside examples of application areas for TinyML-based PdM.

This paper extends the work published in [7]. Table 1 shows how this extension expands on the prior work.

The remainder of the paper is organised as follows: Section II reviews the relevant literature on TinyML and PdM surveys. The parts of the TinyML stack as seen in Figure 1 are reviewed through Sections III to VII. In section VIII, we discuss promising avenues for future research in the field of TinyML-based PdM. Finally, Section IX concludes the paper.

II. RELATED WORK

While the TinyML field is still in its infancy, it has received significant attention in recent years. As shown in Table 1, many papers concentrate on specific application areas of TinyML. For instance, [16], [26] explore its application in healthcare, while [18], [24] investigate the potential positive impact of TinyML on environmental issues. [10] focuses on its use in anomaly detection, and [15] examines the challenges encountered when deploying TinyML in Africa. On the other hand, several other review papers adopt a more general perspective on the field [13], [14], [17], [19], [20], [21], [25], [28], [29], [30], with some taking a systematic approach to their reviews. Notably, [21], [29] provide relatively comprehensive overviews of the TinyML stack, although they pay less attention to the data component. In [29], the authors identify three primary workflows for TinyML solutions: a machine learning-oriented workflow, a hardware-oriented workflow, and a combined workflow that incorporates aspects of both. They further explore optimization techniques and model types and conclude with a discussion on various hardware options and toolchains. Similarly, [21] reviews comparable topics using a different taxonomy while also expanding the scope by covering four specific application areas of TinyML: environment, healthcare, smart farming, and anomaly detection. Despite the existence of all these papers, at this time, no survey has, in our opinion, holistically reviewed TinyML for PdM.

It can, therefore, be challenging to use previous survey literature for TinyML-based PdM, as it tends to focus on standard supervised learning techniques and applications that differ from PdM in especially two areas. Firstly, PdM datasets are often imbalanced or unlabeled, in which case one must often resort to unsupervised learning techniques. Many unsupervised learning models have different characteristics than supervised learning models and require other optimisations. Secondly, PdM data is often of poor quality, which increases the need for data-centric techniques to mitigate the effect of this.

While not strictly surveys, a few books have been written on TinyML. ''TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers'' [31] is a tutorial for deploying several ML applications on ARM-based microcontrollers. The book contains a comprehensive overview of the background of TinyML and Optimisation techniques.

''The TinyML Cookbook'' [32] is another book about TinyML that focuses heavily on practical example applications. Each chapter, except for the two introduction chapters, is dedicated to an application area and a recipe for developing a TinyML solution for that application area.

''Machine Learning Systems with TinyML'' [33] is a community-driven book on everything from the fundamentals of ML systems to advanced topics such as responsible and sustainable ML. This book contains fewer practical examples than other TinyML books, leaving space for a more comprehensive overview of the general field.

III. HARDWARE

At the bottom of the TinyML stack, we find the hardware on which the rest of the system is built. So far, we have described how TinyML systems have few resources. These

TABLE 1. TinyML surveys (✓ = Covered in survey | ~ = Partially covered in survey | ✗ Not covered in survey).

Name	Hardware	Toolchain	Model	Data	Predictive Maintenance	Publication
Tinyml-enabled frugal smart objects: Challenges and opportunities [8]	✗	✓	✗	✗	✗	2020
Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey [9]	✓	✗	✓	✗	✗	2020
Anomaly detection based on tiny machine learning: A review [10]	✗	✗	✓	✗	~	2021
TinyML: Current Progress, Research Challenges, and Future Roadmap [11]	✓	~	✓	✗	✗	2021
Tinyml Meets IoT: A Comprehensive Survey [12]	~	✓	✓	✗	✗	2021
A primer for tinyml predictive maintenance: Input and model optimisation (prior work) [7]	~	~	✓	~	✓	2022
A review of TinyML [13]	✗	✗	✗	✗	✗	2022
A review on TinyML: State-of-the-art and prospects [14]	✓	✓	✓	✗	✗	2022
TinyML in Africa: Opportunities and challenges [15]	✗	✗	✗	✗	✗	2022
A Review of Machine Learning and TinyML in Healthcare [16]	~	~	~	✗	✗	2022
TinyML: A Systematic Review and Synthesis of Existing Research [17]	~	~	✗	~	✗	2022
How TinyML Can be Leveraged to Solve Environmental Problems: A Survey [18]	~	✗	~	✗	✗	2022
Tiny Machine Learning for Resource-Constrained Microcontrollers [19]	✗	✓	✓	✗	✗	2022
Machine Learning for Microcontroller-Class Hardware: A Review [20]	✗	✓	✓	✗	~	2022
A Comprehensive Survey on TinyML [21]	✓	✓	✓	✗	~	2023
Intelligence at the Extreme Edge: A Survey on Reformable TinyML [22]	~	✓	✓	✗	~	2023
Software Engineering Approaches for TinyML based IoT Embedded Vision: A Systematic Literature Review [23]	✓	✗	✗	✗	✗	2023
A Systematic Literature Review of TinyML for Environmental Radiation Monitoring System [24]	~	✗	✗	✗	✗	2023
Tiny Machine Learning: Progress and Futures [25]	✗	✓	✓	✗	✗	2023
TinyML applications and use cases for healthcare [26]	✗	✗	✗	✗	✗	2024
TinyML for low-power Internet of Things [27]	~	✗	✗	✗	✗	2024
A Review on the emerging technology of TinyML [28]	✓	✓	✓	✗	~	2024
A Machine Learning-Oriented Survey on Tiny Machine Learning [29]	✓	✓	✓	✗	✗	2024
TinyML Applications, Research Challenges, and Future Research Directions [30]	~	✗	✗	✗	✗	2024
A Holistic Review of the TinyML Stack for Predictive Maintenance (this work)	✓	✓	✓	✓	✓	2024

resources refer to the various resources provided by the hardware on which the TinyML system is running. It can refer to the computational power of the system, the amount of memory and storage available in the system or even the sensing capabilities that the hardware provides. In this section, we take a deep dive into each of these resource categories and describe how each can be optimised for TinyML.

Apart from the hardware resources available to the TinyML system, an additional interesting aspect of the hardware is its power source. For TinyML devices, this can be anything from a coin-sized battery, the power grid or even an energy harvesting device. In the latter part of the section, we give examples of devices commonly used in TinyML and how they relate to the previously described aspects.

A. COMPUTE

Choosing the computing unit to process TinyML workloads is not as straightforward as it may seem. Many desirable aspects, such as energy efficiency, flexibility, usability, and cost, are often not achievable all at once, and an ideal trade-off must be found for each application.

Below, we list typical aspects that are taken into account when selecting a computing unit for a TinyML system. Afterwards, we discuss common classes of computing units and the trade-offs they generally make between the presented aspects.

Energy Efficiency: A defining characteristic of TinyML hardware is its energy efficiency. Without being energy efficient, TinyML hardware could not be deployed in many areas where a steady power supply is unavailable.

Flexibility: Another important characteristic is the hardware's flexibility, as the wider ML field is rapidly inventing new models. Consequently, state-of-the-art models today may be outdated in just a few years. Having a hardware platform that is inflexible to change will mean that the platform will have to be replaced when new models are needed.

Usability: An often-overlooked and hard-to-measure aspect of TinyML hardware is usability, i.e., how easy it is to use it. Great or poor usability often means the difference between a successful innovation and a forgotten academic exercise.

Cost: An essential aspect of TinyML Hardware is its cost. Embedded systems are generally deployed in huge quantities; therefore, each unit must be cheap to deploy.

1) GENERAL-PURPOSE MICROCONTROLLERS

To a large extent, because of their flexibility, ease of use, and cost, the current TinyML landscape is dominated by general-purpose microcontrollers containing a tiny Central Processing Unit (CPU). This CPU is often based on an ARM Cortex-M architecture. ARM is a large provider of processor Intellectual Property (IP), and their Cortex-M architecture is

their most energy-efficient processor architecture [34], which makes it a natural choice as a general-purpose processor for TinyML.

The ARM Cortex-M architecture contains several variations, ranging from the smallest Cortex-M0 to the Cortex-M7, with the highest performance of any Cortex-M processor. Only some Cortex-M processors support computations using floating point units [34], which are frequently used in traditional ML and NNs. We discuss techniques to run NNs in devices without floating point units in Section V.

Apart from their Cortex-M architecture, the Cortex-A architecture from ARM is also often used in TinyML devices. This architecture from ARM is more powerful but has a significantly higher energy consumption than the Cortex-M architecture [34]. Other non-ARM-based microcontrollers are also in use. For example, many entries in the ESP32 microcontroller lineup use a CPU based on Xtensa IP [35]. Others use a CPU based on the RISC-V open-source processor architecture [36].

TABLE 2. Comparison of computing unit archetypes.

Archetype	Description	Characteristics
CPU	A general purpose processor	Low Energy Efficiency High Flexibility Easy to Use Low Cost
DSP	A processor optimised for processing certain operations at high speeds and efficiency	Medium Energy Efficiency Medium Flexibility Hard to Use Medium Cost
FPGA	A processor with reprogrammable logic gates, i.e., reprogrammable hardware	High Energy Efficiency High Flexibility Very Hard to Use High Cost
ASIC	A processor dedicated to a specific application	Very High Energy Efficiency Little to No Flexibility Hard to Use Medium Cost
GPU	A processor originally dedicated to processing computer graphics	Medium Energy Efficiency Medium Flexibility Challenging to Use Medium Cost

2) HARDWARE ACCELERATORS

When an application demands better energy efficiency or higher throughput than what can be provided by general-purpose microcontrollers, hardware accelerators come into play. Such Hardware accelerators come in many forms, from more flexible Digital Signal Processor (DSP) and Field Programmable Gate Array (FPGA) architectures to custom-made Application Specific Integrated Circuit (ASIC) architectures. See Table 2 for a quick comparison between computing units.

Digital Signal Processors (DSPs): These are specialized processing units that excel at processing multiple data points in parallel in a computing method known as Single Instruction Multiple Data (SIMD). Processing multiple data points in parallel allows a DSP to reach significantly higher throughput than a general-purpose CPU. As a trade-off for being able to process data in parallel, DSPs usually only supports a limited set of instructions compared to a general-purpose CPU. For ML workloads, this is not usually a problem, as many ML algorithms rely on few instructions during execution, for example, Multiply-Accumulate (MAC) instructions for NNs. The increased parallelism in DSPs also increases programming complexity, making the platform significantly harder to use effectively.

Field Programmable Gate Arrays (FPGAs): These are Integrated Circuit (IC) that can be reconfigured after manufacture. This reconfigurability allows for the creation of heavily specialized architectures while maintaining flexibility in the face of changing ML workflows. Using FPGAs comes with some amount of Non-Recurring Engineering (NRE) costs associated with developing the hardware description code for reconfiguration. FPGAs are typically harder to use than both CPUs and DSPs as they require both hardware design specialists and engineers capable of programming customized hardware. Microsoft has been deploying FPGAs in its data centres to accelerate ML workloads [37].

Application Specific Integrated Circuits (ASICs): This hardware provides the fastest and simultaneously least flexible hardware acceleration. An ASIC is custom-designed for a specific application and sacrifices much flexibility to achieve the highest possible performance. ASICs, like FPGAs, are harder to use than CPUs. Because of its highly specialized nature and the long supply chain of manufacturing IC, the NRE costs of designing ASICs are significantly higher than other platforms. Experience has shown that designing and manufacturing ASICs often takes more than two years [33]. Such a long product development time and the inherent inflexibility of ASICs can mean that an ASIC can be outdated before it is fully developed. Developing an ASIC is, therefore, a costly and risky venture. An organization does, however, not necessarily need to develop its own ASIC. Many ASICs have already been developed for general ML systems. Some are available as commercial products and thus allow third parties to achieve the performance of ASICs without the associated NRE costs. The ARM Ethos product line [38], Google Coral product line [39] and the Renesas DRP-AI [40] contains examples of TinyML (although at very different power levels) devices using an ASIC to accelerate the execution of NN based models.

Graphics Processing Units (GPUs): In traditional ML systems, GPUs play an important role, where the ML models are often trained and executed on GPUs. In the TinyML space, Nvidia has launched the Jetson Nano product line [41] of low power GPUs for executing edge

ML workloads. At a power consumption of 5-10 Watts, according to official figures, it is debatable whether this can be considered a TinyML device.

Researchers are currently investigating even more ways to accelerate the computation of ML models. Particularly promising avenues include near- and in-memory, neuromorphic, approximate, and analogue computing. The basic idea behind near- and in-memory computing is that memory latency is the most significant bottleneck for ML hardware performance. To reduce this latency, computation and memory units are moved as close as possible [42]. Neuromorphic computing hardware attempts to replicate the computational structure of the human brain. By doing so, it aims to achieve computational efficiency unachievable by traditional hardware architecture [43]. Approximate computing is a computing paradigm that accepts some degree of error during computation to reduce the computational complexity of a task [44]. Analogue computing abandons the digital representation of values common in contemporary hardware to speed up computation [45]. These new computing technologies are not mutually exclusive and are often combined, e.g., neuromorphic hardware may be implemented using analogue circuits.

Comparing TinyML devices can be tricky, and lofty performance promises made by manufacturers often fail to deliver in the real world. To enable apples-to-apples comparisons of TinyML hardware, researchers have created the MLPerf Tiny benchmark. In this benchmark, the speed and energy efficiency of participating TinyML devices is recorded while executing certain models [46], [47].

B. MEMORY

The choice of memory, like the choice of hardware computing unit, requires balancing competing parameters. While there are many more, we will describe some of the more important parameters in this subsection.

Memory capacity: The memory capacity of TinyML devices is generally low. Typically, these devices have less than 1 MB of low latency volatile memory and are in the low MB ranges of higher latency persistent memory. There are, however, TinyML devices that go far lower than this boundary. The memory capacity is a significant decider for which ML models it is possible to deploy on TinyML devices.

Latency and Throughput: Different memory technologies and their proximity to the computing unit can have greatly different latency and throughput. TinyML hardware, which includes hardware accelerators, often requires low-latency high-throughput memory to achieve the accelerator's full performance. Low latency and high throughput are often achieved by combining caches and on-chip Static Random-Access Memory (SRAM) with high bandwidth interconnect close to the hardware compute units. Latency and throughput are typically less important for persistent memory.

Power Consumption: As with the computing unit, the power consumption of the memory must also be kept low. One way this is achieved is to use SRAM instead of Dynamic Random-Access Memory (DRAM), which has a higher cost but a lower idle energy consumption. Similarly, flash memory also consumes less power than most other persistent memory technologies.

Cost: To create affordable TinyML systems, the cost of memory units must also be kept low. This can be achieved in several ways, e.g., by reducing memory capacity or using cheaper DRAM instead of SRAM.

C. SENSING CAPABILITIES

Many TinyML applications involve sensing the environment in which the TinyML system is placed and employing a TinyML model to understand the sensed data. For this reason, some TinyML hardware contains built-in sensors. Others rely on interfaces like General Purpose Input/Output (GPIO) pins to connect to various sensors.

Built-in sensors may provide greater reliability, cost and usability - however, they sacrifice the flexibility and repairability of external sensors.

Whether the TinyML hardware relies on internal or external sensors, the sensors must live up to the application's requirements. The most common parameters to set requirements for include accuracy, sampling frequency, response time, range, sensitivity and reliability.

Accuracy: The maximum difference between the actual real-world value of the sensed parameter and what is recorded by the sensor.

Sampling Frequency: How often a sensor can capture data.

Response Time: how long it takes for a sensor to record that a change has happened in its environment.

Range: The range of values a sensor can capture about its environment. For example, a temperature sensor may have a minimum and maximum temperature that it can record.

Sensitivity: How small a change in its environment that a sensor can recognize.

Reliability: How probable it is that a sensor will perform its functionality without failure. This can also include which conditions the sensor can operate in.

D. ENERGY SOURCE

A TinyML device can be powered in several ways. The most straightforward way is to power the device by directly connecting it to the power grid. This way, the device can be powered and work at full capacity as long as the power grid supplies power. Directly connecting the TinyML device to the power grid naturally requires that such a connection is possible. For example, if a TinyML device is to be deployed in a remote area or constrained space, such as a connection may not be feasible.

A common alternative is to run the TinyML device on battery power. In this case, there is no need for a power line,

and this configuration can thus be applied almost everywhere. Batteries, however, have the downside that they can only supply a limited amount of energy before needing recharging or even replacement. Deploying battery-powered TinyML devices on a large scale can, therefore, require significant manual work to maintain batteries.

A less common alternative is to power the TinyML device with an energy harvesting system. An energy harvesting system is a system that harvests energy from its environment – for example, a solar panel. One of the many more creative examples is to power a TinyML device through photosynthesis [48]. Combining a TinyML device and an energy harvesting system can avoid the downsides of both a directly connected and a battery-powered system. Unfortunately, predicting the energy available in a system powered by energy harvesting is challenging. Therefore, the system must use the energy available intelligently to fulfil its requirements. To our knowledge, little research has considered using energy harvesting systems for TinyML.

E. SUSTAINABILITY

As we shall discuss in Section VII, TinyML can contribute to the United Nations (UN) sustainability goals in many ways. Creating TinyML systems, however, has a non-negligible impact on the climate and environment. These impacts have been shown to be dominated by the production of TinyML devices and their batteries [49].

Because of this, it is paramount for sustainable TinyML to limit the production of new TinyML devices and batteries to what is ultimately necessary. Flexible TinyML Hardware can help alleviate this problem, as inflexible hardware may have to be replaced with new hardware when changes occur in other parts of the TinyML stack. If possible, TinyML systems that are directly connected to the power grid or use energy harvesting systems will also be more sustainable as they reduce the demand for batteries.

F. EXAMPLES OF TinyML HARDWARE

There are a few devices that often appear in TinyML publications and projects. We list and describe some of them here. Note that all systems can use whatever energy source can provide their required power.

Arduino Nano 33 BLE Sense: Arguably, the most commonly used device is the Arduino Nano 33 BLE Sense. This device features an ARM Cortex-M4 processor with support for floating-point operations. In terms of memory, the device has 256 kB of SRAM and 1 MB of persistent flash storage memory. It also includes a range of internal sensors to sense proximity, temperature, motion, humidity, pressure and audio. The device does, however, not feature a camera, so it requires an external sensor for visual sensing [50].

Sparkfun Edge: Another popular device is the Sparkfun Edge. This device was designed in collaboration between its producer, Ambiq and TinyML engineers [31]. Like the Arduino, this device also uses an ARM Cortex-M 4 processor with support for floating-point operations [51]. With 384 kB of SRAM and 1 MB of flash storage, the memory characteristics of this device are also close to that of the Arduino. The device has a built-in camera, microphone and accelerometer [31].

ESP32-C6-DevKitC-1: This device is a part of the ESP32 series of low-power microcontrollers. The microcontroller does not use a processor from the ARM Cortex-M series as the previously discussed devices and instead uses two RISC-V processors. It contains modules for Wi-Fi, Bluetooth Low Energy (BLE), Zigbee and Thread, plus a temperature sensor [52]. With 512 kB of SRAM and 8 MB of flash storage, this device can be used to deploy models larger than the previously mentioned devices.

Google Coral Dev Board Micro: The Google Coral Dev Board Micro is an example of a TinyML device containing an onboard hardware accelerator, namely the Coral Edge TPU. Alongside the Edge TPU, the board has two Cortex-M processors, a camera and a microphone for general processing and sensing. To store data for this system, the dev board has 64 MB of DRAM and 128 MB of flash memory

Raspberry Pi 4 Model B: The Raspberry Pi 4 Model B uses an ARM Cortex-A72 processor with four cores. This processor also supports floating-point computations. With options for up to 8 GB of DRAM, this device is much more powerful and power-consuming than some of the previous devices. The device has no built-in sensors but provides a wide range of connectors, including GPIO pins for external sensors. Unofficial measurements have reported the power consumption of the Raspberry Pi as around 2.2-4 W [53].

IV. TOOLCHAIN

Several toolchains aim to ease the development of TinyML applications. This section will give an overview of the available toolchains while comparing them according to relevant metrics. Many TinyML toolchains focus solely on NN models. See Table 3 for an overview.

A. NEURAL NETWORK FOCUS

TensorFlow Lite Micro (TFLM): This is an open-source interpreter for NNs whose primary contributor is Google. The interpreter is a lean implementation of the TensorFlow (TF) interpreter, which can execute TensorFlow Lite (TFL) models on microcontrollers. The aim of TFLM is not to provide the most optimized implementation of all kernels but to offer a modular and flexible framework that can be easily enhanced with optimized kernels created by hardware vendors. Many hardware vendors have contributed to this initiative, making TFLM the most widely used TinyML library. Generally, the TFLM library can be compiled

by any C++ compiler that supports C++17. Some of the supported hardware platforms with optimized kernel implementations include ARM Cortex-M-based microcontrollers, Renesas RA microcontrollers, and the ESP32 microcontroller [54].

Edge Impulse: This is a toolchain provided as a Software-as-a-Service model. It was founded in 2019 and is based on TFLM. It has since expanded to support more models and devices. Edge Impulse is an end-to-end toolchain providing a complete pipeline from data acquisition, labelling, model training, optimisation, and deployment. It can either take in a dataset or use a pre-trained model, which can be in formats such as TF, TFL, Open Neural Network Exchange (ONNX), or XGBoost. The toolchain can output a bag of C++ files, library files for popular IDEs, or even compiled binaries that can be flashed onto the device. The whole pipeline is supported by a user-friendly web interface that makes it easy to use, even for non-experts. Even though this platform is not open-source, most of its features are free to use for non-commercial projects [55].

Embedded Learning Library (ELL): This is another open-source library for NNs. Microsoft is the primary contributor to this one. Like TFLM, this library supports running on ARM Cortex-M-based microcontrollers but also the more powerful ARM Cortex-A architecture. The Raspberry Pi is the primary focus of ELL, which also supports several smartphones. The library generates C++ code for microcontrollers based on pre-trained NN models in the Microsoft Cognitive Toolkit (CNTK), Darknet or the ONNX format [8].

ARM-NN: ARM themselves have also released a library for running NNs on microcontrollers. This library only supports ARM Cortex-A, Mali, and Ethos-based processors and is therefore targeted towards more powerful devices than, e.g. TFLM. An operating system is necessary to run the output model, as it cannot operate directly on bare metal [8].

CMSIS-NN: For Cortex-M devices, Arm has released the CMSIS-NN library. The CMSIS-NN library provides efficient implementations of NN functions for ARM Cortex-M-based systems [56].

STM32Cube.AI: Targeted towards its STM32 boards, STMicroelectronics has developed the STM32Cube.AI tool that can be used with STM32CubeIDE to develop TinyML models. This tool can optimally convert TF, Keras, PyTorch, and Matlab models to C code using techniques like quantisation, graph optimisation, and memory optimisation. Compared to TFLM, they claim the possibility of getting up to 60% faster execution and up to 20% save in flash and RAM space. A unique feature that STM32Cube.AI offers is the possibility of benchmarking the models on real STM32 boards available on their board farm. This allows prospective users to find the best hardware for their specific application [57].

Artificial Intelligence for Embedded Systems (AIfES): The Fraunhofer Institute for Microelectronic Circuits and Systems develops AIfES. The library uses a mixed license, which is open-source for private and other open-source projects but proprietary for commercial projects. This library takes Keras or TF models that do not have to be pre-trained and generate NNs compatible with a range of GCC-compatible microcontrollers [8].

NanoEdge AI Studio: Another proprietary tool from STMicroelectronics is the NanoEdge AI Studio. At its core, this tool functions as a search engine for four model types: Anomaly Detection, Outlier Detection, Classification, and Regression. Notably, the anomaly detection model can be further trained directly on the device. The generated models boast a minimal memory footprint, requiring only 1-16 kB of RAM and less than 10 kB of flash memory [58].

Ekkono: Ekkono is a Swedish company that develops a proprietary library for running NNs and traditional ML models on microcontrollers. Among the NN models, the library supports Multi-Layer Perceptron (MLP) for regression problems and can further train them on the device. This toolchain requires the users to generate the model in Ekkono's environment and then deploy it to the microcontroller. The output of the toolchain can be understood by any C compiler that supports C99 [59].

e-AI Translator: While TinyML mainly focuses on ARM-based microcontrollers, manufacturers of non-ARM-based microcontrollers have also developed toolchains for NN inference on their devices. e-AI Translator is an example of this. The library generates C code compatible with most families of Renesas MCUs from TF, Keras, or PyTorch models. Although not open-source, this tool is integrated into the Renesas e2 studio IDE and can be used free of charge [60].

μTensor: μTensor is a library that takes TF models and generates C++ code for microcontrollers. It runs on top of the MBed operating system [61].

TinyMLgen: This library can convert trained TF Lite models to a plain C array for inference in microcontrollers [62].

CMix-NN: Most toolchains so far have been developed by enthusiasts or companies. This library is developed by researchers from the University of Modena and Reggio-Emilia and the University of Bologna. The library is an inference library for running NNs on ARM Cortex-M based systems [63].

FANN-on-MCU: Another library made by a research group is the FANN-on-MCU library. This library takes models made in the Fast Artificial Neural Network (FANN) library and generates code that can run on ARM Cortex-M or Parallel Ultra Low Power (PULP) processors [64].

microTVM: The microTVM library extends the Apache TVM NN model deployment framework. It allows deployment of, e.g., Tensorflow, PyTorch, and ONNX models onto bare-metal microcontrollers. Its design principles ensure that this library remains suitable for a wide range of devices, offering numerous optimisations while maintaining control and flexibility. However, operating microTVM may require a more advanced user expertise [65].

TinyEngine: This inference library was proposed alongside TinyNAS to design tiny but accurate models. Contrary to TFLM and CMSIS-NN, which work using an interpreter, TinyEngine creates compiled models that only ship code that the model will execute. TinyEngine also uses an adaptive memory scheduling that seeks to fully utilize its given memory throughout its computation [66]. To our knowledge, it supports executing models generated in TF Lite and PyTorch together with tinyNAS.

Neurona: This library is specific to Arduino and allows deploying Artificial Neural Networks (ANNs) to Arduino boards. The library has been tested on the Arduino Uno and Arduino Mega boards [67].

B. TRADITIONAL MACHINE LEARNING FOCUS

MicroMLGen: While most toolchains focus entirely on NNs, this library brings many traditional ML algorithms to microcontrollers. The library ports models created in Scikit-learn [68]. It primarily focuses on porting models to the Arduino platform but creates C code that could be compatible with other platforms [69].

sklearn-porter: This is another library that enables porting traditional ML models created in Scikit-learn to microcontrollers. It is not only focused on porting models to microcontrollers but also to web and desktop applications. Apart from converting models to C code, the library can also convert models to Java, JavaScript, Go, Ruby and PHP. The library has not yet had its first major release, so some estimators are unavailable for some target languages [70].

m2cgen: Another alternative for porting Scikit-learn models to microcontrollers is m2cgen. Like sklearn-porter, this library converts models into native code, which enables it to run on microcontrollers. It supports more languages and models than sklearn-porter [8].

weka-porter: This library is a sister library to sklearn-porter. As the name implies, this library ports models from the Weka library instead of the Scikit-learn library. It supports porting decision trees to C, Java or Javascript [71].

EmbML: This library converts Scikit-learn or Weka models into C++ or C code. This library, contrary to, e.g. sklearn-porter or m2cgen, focuses on making the converted models run in resource-constrained hardware [72].

Reality AI: Acquired by Renesas, Reality AI is an end-to-end toolchain designed to take in data and generate C or C++ code compatible with all Renesas microcontrollers. By extracting hundreds of feature sets and testing numerous models, Reality AI identifies the optimal pairs for the given data, presenting users with detailed information to help them select the best option for their application [73].

emlearn: The last library that we look at supports both NNs and traditional ML models. It can port Scikit-learn or Keras models to microcontrollers. It supports common tree-based models, basic sequential NNs, Naive Bayes, and Elliptic Envelope [74].

These projects vary significantly in size. TFLM and ELL, for example, is created by two major Information Technology (IT) companies. Other projects are developed by research groups and small or medium-sized companies. Finally, some projects are not much more than a hobby project for a developer. This discrepancy should be considered when deciding on the library to use for a project.

Note that several of these projects overlap and have a similar focus and that some toolchains use other toolchains internally E.g. TFLM uses the CMSIS-NN library in its interpreter.

Many of the described toolchains follow an interpreter-based approach, in which an interpreter reads and executes a binary representation of the model. This approach makes the development of the platform and the model execution easier and more flexible. It also allows for the hot swapping of models, as only the model representation needs to be replaced, not the entire firmware. The downside of this approach is that it is usually slower than a compiled model and takes up more memory.

An optimisation unique to the interpreter-based execution of ML models is to remove unnecessary operations from the interpreter. This leads to a smaller interpreter size in a TinyML system [54]. For example, even though Edge Impulse is based on TFLM, it manages the required operations automatically to minimize them and lower the memory footprint. Its EON compiler goes further by integrating the interpreter into the code, shifting the platform's behaviour from an interpreter-based approach to a compiled model-based approach.

Most of the toolchains mentioned in this section do not support on-device training and, therefore, expect a fully trained model to be ported onto the TinyML hardware. Without on-device training, models can not adapt to changes in their environment. However, finding the extra resources required for on-device training in typical TinyML hardware can be challenging. Likewise, getting the feedback required for supervised learning on the fly can be challenging.

Researchers have recently focused on improving this situation. A k-Nearest Neighbor (KNN) model can implement on-device training by simply storing inputs in an on-device database and using these saved inputs to cluster future inputs.

TABLE 3. A list of TinyML toolchains. Adapted and updated from [8]. See the reference next to library names for the source of each library.

Name	Models	Platform	Related Toolchains	Open Source	On-Device Training
TFLM [75]	NNs	Multiple	TF	✓	✗
Edge Impulse [55]	Mix	Multiple	TFLM, TF, ONNX, XGBoost	✗	✗
ELL [76]	NNs	ARM Cortex-M, ARM Cortex-A	CNTK, Darknet, ONNX	✓	✗
ARM-NN [77]	NNs	ARM Cortex-A, ARM Mali, ARM Ethos	TF, Caffe, ONNX	✓	✗
CMSIS-NN [78]	NNs	ARM Cortex-M	TF, PyTorch, Caffe	✓	✗
STM32 Cube.AI [57]	NNs	STM32	Keras, TF Lite, Scikit, ONNX, Matlab, PyTorch	✓	✗
AIfES [79]	NNs	GCC compatible	TF, Keras	Mixed	✓
NanoEdge AI Studio [58]	NNs	ARM Cortex-M	-	✗	✓
Ekkono [59]	NNs	Multiple	-	✗	✓
e-AI Translator [60]	NNs	Renesas Microcontroller Units (MCUs)	✗	✗	
µTensor [61]	NNs	Mbed OS	TF	✓	✗
TinyML gen [62]	NNs	Multiple	TF Lite	✓	✗
CMix-NN [80]	NNs	ARM Cortex-M	MobileNets	✓	✗
FANN-on-MCU [81]	NNs	ARM Cortex-M, PULP	FANN	✓	✗
microTVM [65]	NNs	Multiple	TF, PyTorch, ONNX	✓	✗
TinyEngine [82]	NNs	Multiple	TF Lite, PyTorch	✓	✗
Neurona [67]	NNs	Arduino	-	✓	✗
MicroMLGen [69]	Traditional ML	Arduino	Scikit-learn	✓	✗
sklearn-porter [70]	Traditional ML	Multiple	Scikit-learn	✓	✗
m2cgen [83]	Traditional ML	Multiple	Scikit-learn	✓	✗
weka-porter [71]	Traditional ML	Multiple	Weka	✓	✗
EmbML [84]	Traditional ML	Multiple	Scikit-learn, Weka	✓	✗
Reality Artificial Intelligence (AI) [73]	Traditional ML	Multiple	-	✗	✗
emlearn [85]	Mix	Multiple	Scikit-learn, Keras	✓	✗

As each input needs to be saved on the TinyML device for future inference, on-device training for KNN models can consume a significant amount of memory [22].

For NNs, some research has suggested adding additional output layers to the models. These layers are placed downstream from the original output layer of the model. New layers are necessary as the existing NNs are typically deployed as a frozen graph that cannot be further modified. The TinyML on-device training consists of training these additional output layers online to adapt to changes in the environment [86], [87]. It is interesting to note that this additional layer will not consume much memory on a standard classifier. However, it may be infeasible to add an additional full-size output layer to, e.g. an autoencoder, as its original output layer is already the size of the input layer. Other approaches suggest freezing NN weights and only update biases, which removes the need to store intermediate activations [88]. Further innovative approaches like quantisation-aware scaling or sparse update strategies [89] have been proposed to increase training effectiveness and reduce the memory footprint of on-device training. Still, the field of on-device training is in its infancy, and more research can be expected in the future.

V. MODEL

At the third layer of the TinyML stack, we find the ML models themselves. These models process captured data to provide intelligent predictions. Years of ML research have produced many more models than what can be covered in a single review. Therefore, we will only cover more widely

used models, with a particular emphasis on models used for PdM systems.

A. MODEL TYPES

ML models for PdM are typically either supervised or unsupervised. Supervised models can be further distinguished by the type of data which they output. Models that output categorical values accomplish a task known as classification, whereas models that output a continuous value accomplish a task known as regression. Other models are unsupervised and can be trained on raw input data without specifying the desired output.

Amongst the simpler ML models used for PdM, we see linear regressions, decision trees, random forests, Support Vector Machines (SVMs), KNNs.

Linear Regression: This model fits a linear equation between input data and the value being predicted. A linear regression model is a supervised ML model and is, as the name implies, primarily used for regression.

Decision Tree: This model sets up a tree of decision nodes. A prediction by a decision tree is made by reaching a leaf node in the decision tree, each containing a prediction. To reach a leaf node, a data sample is queried at nodes throughout the tree, which makes a decision about which child node to evaluate next based on the data sample. A decision tree model is a supervised ML model which can be used both for classification and regression tasks.

Random Forest: This model is made up of several decision trees. The random forest model makes predictions based on the participating decision trees' consensus or average value. A random forest model is a supervised ML model which can be used for both classification and regression tasks.

Support Vector Machines (SVMs): This model makes predictions by constructing hyperplanes in a high-dimensional space that separates data classes by the largest possible margin. A SVM is a supervised ML model for classification. A variant of the SVM called Support Vector Regression can be used for regression tasks.

K-Nearest Neighbor: This unsupervised learning model makes predictions for a data sample based on its k nearest neighbours - typically measured by the Euclidian distance. The model can be used both for classification and regression tasks.

The simplicity of these models is beneficial in TinyML systems, as the models consume little resources to run.

Various versions of NNs often perform better for more complicated tasks. The basic variant of a NN is the MLP, which consists of layers of several neurons, with all neurons of neighbouring layers being connected. Alternative variants are Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and transformers consisting of convolutional layers, recurrent connections, and attention heads, respectively. Convolutional layers train a filter to pass over a tensor, usually an image, to extract features that help later classification/regression. Recurrent connections are connections in NNs propagate from one input sample to the next. Attention heads are special neural network structures that imitate natural cognitive attention [90]. Many other specialised variants exist. One that is particularly interesting for PdM, and more specifically anomaly detection, is the autoencoder. The autoencoder NN takes an input, compresses it, and reconstructs it. By training on normal samples, it learns to reconstruct normal samples. It will, however, not be able to reconstruct anomalous samples that it was not trained on - thus making it usable as a model for anomaly detection.

Especially CNNs have seen a lot of TinyML research to improve their efficiency. Here, we chronologically summarize the main contribution of important CNN models to the field of TinyML.

SqueezeNet: This network introduces fire modules to achieve a more efficient architecture. Each fire module consists of a squeeze layer, which employs 1×1 convolutions, and an expand layer, which incorporates both 1×1 and 3×3 convolutions. The squeeze layer reduces the number of input channels, allowing the expand layer to process them with lower computational cost and fewer parameters. To maintain high network performance, it is recommended to downsample the feature maps later in the network. This will increase the computational load while keeping the number of parameters constant [91].

MobileNetV1: This network by Google proposed a new CNN architecture optimized for mobile and embedded devices. The key innovation of this architecture is the replacement of standard convolutional layers with depthwise separable convolutions, reducing the number of parameters and the computational cost of the network. Depthwise separable convolutions consist of two stages: first, a depthwise convolution applies a single convolutional filter to each input channel individually; second, a pointwise convolution uses a 1×1 convolution to combine these channels into new feature maps. This architecture has been refined and employed in several other networks, such as MobileNetV2 and EfficientNets [92].

MobileNetV2: This network improves upon MobileNetV1 by introducing inverted residual blocks. These blocks are similar to the depthwise separable convolutions used in MobileNetV1 but are preceded by a 1×1 convolutional layer. This layer merges information and increases the number of channels, enabling the block to capture richer features. Inspired by the ResNet architecture, a skip connection is added to the block to enhance gradient flow through the network [93].

MNasNet: With the spread of Neural Architecture Search (NAS), researchers acquired a new method for designing CNNs. MNasNet is one of the earliest attempts to bring NAS to the realm of TinyML. Their reinforcement learning search algorithm incorporates a combined

loss function that includes the network's latency as a factor. Instead of using a proxy for latency, they perform real-time latency measurements on a Pixel phone to guide the search. Another novelty of this work is dividing the network into several blocks, allowing the algorithm to search for the optimal architecture for each block separately. This balance enables the search algorithm to significantly modify the network's architecture while maintaining a manageable search space [94].

EfficientNet: This network proposes a compound scaling method that uniformly scales the network's depth, width, and resolution. By systematically scaling these dimensions, they demonstrated better performance on architectures such as ResNet and MobileNets. Leveraging NAS, they introduced EfficientNet-B0, the baseline model for the EfficientNet family, which is more efficient than previous state-of-the-art networks [95].

Once For All (OFA): The novel idea behind this network design is that a single large network can be trained to encompass many subnetworks, each optimized for different properties such as accuracy, latency, or model size. Users can select the subnetwork that best fits their specific application requirements, whether it be a microcontroller, a smartphone, or even a server. This network achieved new state-of-the-art results on the ImageNet dataset with fewer than 600M MACs [96].

Others: While we have aimed to cover the most important CNN models for TinyML, many other models have been proposed. Some of these include ProxylessNAS [97], MobileNetV3 [98], and MCUNets [66], [99]. The full study of the evolution of CNN models can be a topic for a separate review.

B. CHOOSING A MODEL

The choice of model for TinyML-based PdM depends on the hardware and data available for the PdM application. If the hardware resources are extremely limited, then a simple ML model could be the right choice. In more powerful systems, NNs might be the better choice, especially for complicated tasks such as image processing.

Supervised learning approaches are likely the best choice if the data includes impending failure labels or RUL labels. Imbalanced data can, however, be an issue for supervised learning approaches. If the imbalance is limited, over- or under-sampling techniques can synthesise a balanced dataset. Some literature also proposes to use generative models or transfer learning to solve these problems [2].

If the data is unlabeled or extremely imbalanced, we must turn to unsupervised learning. For PdM, this usually means that we want to do anomaly detection. There are a few ML models that are suitable for anomaly detection. Two of the most popular models are KNN and autoencoders [2], [100]. A KNN model is a traditional ML model, which clusters observations based on features derived from them. The idea is that an anomalous sample will diverge from the cluster(s) of normal observations and that it can thereby be identified as an anomaly. The autoencoder model is explained earlier in Section V. For both models, a loss threshold should be set for classifying a sample as normal or abnormal [2].

As with other NNs, we can introduce convolutional layers to autoencoders to improve their capabilities in image processing. In this case, we call the model a convolutional autoencoder.

C. MODEL OPTIMISATION

The disadvantages of TinyML listed in Section I most stem from microcontrollers being less powerful than larger computers. Therefore, it is natural to apply optimisations to a TinyML system to make it consume fewer resources. Note that these optimisations often lead to a reduction in accuracy compared to full-blown models. Therefore, which optimisations to apply to a system concerns finding the right balance between multiple objectives. As explained in section IV, the focus of TinyML research has primarily been on optimising NN models. NNs typically also require more compute and memory than traditional ML models. Therefore, this section will focus on optimisations for these models. Overall, we describe nine ways to optimise the performance of NN-based models for TinyML. These are quantisation, pruning, clustering, neuron merging, knowledge distillation, cascading architectures, early exit networks, Automated Machine Learning (AutoML), and compression.

1) QUANTISATION

By default, most NN toolchains represent their weights, biases and activations as 32-bit floating-point values. This default poses two problems when deploying their models on microcontrollers. Firstly, not all microcontrollers have hardware support for floating-point units, in which case floating-point operations are emulated in software. Such emulation can cause significantly slower processing. Secondly, the many 32-bit values can take up a large part of the memory of microcontrollers. For example, the Arduino Nano 33 BLE Sense [50], mentioned in Section III, has 256 kB of RAM. That leaves room for 64.000 weights, biases, and activations. While that might sound like a lot, many modern NNs have much more. E.g. AlexNet and Resnet-50 both contain more than one million weights, biases and activations [101]. That is, without considering the memory required for the model structure, input data, and application.

Fortunately, research has shown that quantising NNs is possible while retaining a good model [102]. This ''quantisation'' process concerns two optimisations done simultaneously. The first is to convert the floating-point values into fixed-point values. Fixed point arithmetic is generally less complicated than floating point arithmetic and requires less computation on most devices. The second is to reduce the original bit to a lower bit-width. A lower bit width reduces the memory needed to store each value and

requires less complex computing hardware. Some hardware architectures can perform multiple MAC operations per cycle on low-bit-width data, significantly reducing execution time. Reducing the bit-width to 8 bits is a typical configuration for quantisation, which has experimentally shown to provide a good trade-off between accuracy and resource consumption.

"Uniform affine quantisation" is the most common form of quantisation, also known as asymmetric quantisation [102]. Three parameters parameterise this form of quantisation: a scale factor s, a zero-point z and a bit-width b. The scale factor decides the size of a "step" in the floating point domain corresponding to one step in the fixed point domain. The zero point parameter maps the zero point of the floating point domain to the fixed point domain. This parameter is needed to ensure that common operations, e.g. zero padding or Rectified Linear Unit (ReLU), do not introduce errors when quantised. Finally, the bit-width decides the size of the fixed point grid. Using these parameters, we can map from a floating point value x to a fixed point value x_{int} using the equation 1:

$$x_{int} = \text{clamp}(\lfloor \frac{x}{s} \rceil + z; 0, 2^b - 1) \tag{1}$$

where clamp is the function described in equation 2:

$$\text{clamp}(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c. \end{cases} \tag{2}$$

A clipping error can occur if x is below 0 or above $2^b - 1$. In this case, the floating point value is mapped to the lowest or highest possible fixed point value, respectively. Increasing the bit-width or scaling factor can reduce the clipping error. Increasing the bit-width will increase memory and compute consumption. Increasing the scaling factor will result in an increased rounding error due to the rounding error being in the range of $[-\frac{1}{2}s, \frac{1}{2}s]$. Therefore, the scaling factor can be modified to achieve a trade-off between clipping and rounding errors [102].

After quantisation, multiplying or adding two fixed points can quickly create an overflow situation. This overflow happens due to the minimum and maximum 32-bit floating-point weights being mapped close to the minimum and maximum values for the fixed point. Consider that applying just the smallest multiplication or addition to the largest 32-bit floating-point weight after quantisation will result in an overflow. Therefore, some approaches only quantise weights and biases (or only weights, as they grossly outnumber biases) and let the remaining activations (and biases) stay as 32-bit floats. A way to quantise activations is to compute the fixed point computations and store the result in higher bit-width fixed points. The result can then be scaled down to the original bit-width for the following computation. Therefore, quantisation can reduce the model size, increase inference speed, and make the model run on a broader range of devices. The downside is a potential loss in accuracy [10]. This potential accuracy loss can be reduced using quantisation-aware training. In this method, a model is trained with the knowledge that it will be quantised later [103]. Note that a four times reduction in the number of weights, biases and activations will make neither AlexNet nor Resnet-50 fit in the Arduino we are considering. To achieve that, we require further optimisations or smaller models.

Researchers have even been looking into binarisation of NNs, which can decrease the size of NNs by up to 32 times and the inference time of the networks by up to 52 times [104]. We refer to [102] for a white paper exclusively about NN quantisation.

2) PRUNING

Pruning is an optimisation which seeks to remove parameters in a NN to make the model more efficient. To do so, pruning typically associates a score to each NN parameter and prunes the parameters with the lowest scores either locally, e.g. per layer, or globally for the whole network. Scoring is often based on the absolute value of a parameter but can also be based on contribution to activations, gradients or even trained importance coefficients [105].

Pruning NN parameters in an unstructured way can often lead to disappointing results, as the resulting NN is sparse. Sparse NNs are not supported by many toolchains and hardware, which can result in slower inference speeds and little to no memory gains. Structured pruning recognizes this flaw and applies pruning in a structured way to ensure that the resulting NN model can still be executed and stored efficiently [106]. Structured pruning approaches can, for example, prune every parameter associated with a specific neuron, group of neurons, filters or channels simultaneously [105].

New neural network accelerators are continuously being designed to leverage more flexible pruning algorithms. For example, NVIDIA's A100 Tensor Core GPU supports 2:4 structured sparsity, which allows two out of every four elements in a weight tensor to be pruned [107].

3) CLUSTERING

An approach that is closely related to both quantisation and pruning is clustering. In this optimisation technique, weights are clustered into groups, where all weights in one group are assigned the same weight. This clustering allows one value in memory to represent several weights. However, each weight with this value needs to store a pointer to the value. Thus, depending on the bit width of the weight values, this optimisation may yield more or less savings in memory. While structured clustering, like structured pruning, can reduce the memory consumption of a model, clustering does not directly speed up computation. However, depending on the hardware, the processor cache may experience fewer misses due to the reuse of weight values. The paper that initially introduced clustering claims that their approach reduced the size of a NN by 27 to 31 times [108]. This reduction comes after pruning has reduced the size by 9 to 13 times, as reported in Section V-C2.

4) NEURON MERGING

In an MLP, many neurons within the same layer of a trained model often learn similar features [109]. By merging these neurons, redundancy can be eliminated, thereby reducing the model size and computational load [110]. This technique also applies to other network types such as CNNs. Unlike clustering or unstructured pruning, neuron merging directly impacts performance without the need for additional software or hardware support [111].

5) KNOWLEDGE DISTILLATION

Larger NN models often perform better than smaller models on complicated datasets. However, deploying such large models on TinyML devices might be infeasible. A solution is to ''distil'' the knowledge of a large model into a smaller model, which, due to its reduced resource consumption, can be deployed on TinyML devices.

This technique is known as knowledge distillation. The idea behind this method is to train a large model and then use the predictions generated by this model to train a smaller model. Thus, the small model trains both using the ground truths and the predictions generated by the large model [112].

Consider the following example. We want to create a small model that can classify the contents of an image. Normally, we train the network to make the same classification as the ground truth labels. In knowledge distillation, we first train a larger model on our data. We then have the larger model classify all images in the dataset. Then, we train the small classifier not just to classify the ground truth but also to make similar classifications as the larger model. This training can be achieved by altering the loss function of the smaller model. Often, the larger model is referred to as the ''teacher'', and the small model as the ''student''.

As knowledge distillation aims to remove parts of a larger model that is not necessary to achieve some level of predictive performance, it is closely related to ablation studies. In ablation studies, tests are carried out on which model components are necessary for the model's predictive performance and which components can be left out to reduce model complexity and resource consumption.

6) CASCADING ARCHITECTURES

In some cases, it may be infeasible to deploy a desired model in a TinyML device due to not meeting the constraints of the device or the use case. In these situations, it may be beneficial to introduce smaller and less resource-consuming models that decide when to engage the main model.

If the original model was infeasible for the TinyML device due to its energy consumption, then deploying the models onto the device in unison may be possible. This possibility is due to executing the original model less frequently and thus consuming less energy. The same could be true for a model executing too slowly For example, a long processing time may be okay in some cases but not for every input.

Another option is to deploy the smaller model on the TinyML device and the original model on a networked device. Note that this compromises some of the advantages of TinyML listed in Section I.

The latter option is, for example, used to power the Google Assistant on many Android phones. In this system, a small model runs locally on the smartphone, which listens for the ''Hey Google'' keywords. Once it detects these keywords, it sends the remaining speech to a larger model in the cloud to further process the request [31]. While cascading architectures are usually restricted to model type and structure, research has looked into a cascading use of internal hardware in a system [113].

7) EARLY EXIT NETWORKS

Early exit networks are NNs that provide some means for inference to return a result without computing every layer. Executing some parts of the model will not be needed in case of an early exit, leading to a reduced inference time. However, adding early exits to NNs can affect the final accuracy, as some early layer weights adapt to the early exit instead of the final exit. Thus, adding early exits to a NN results in a tradeoff between inference time and accuracy. See the following research paper that also proposed early exit networks for TinyML for a thorough discussion of how to alleviate this effect [114].

8) AutoML

Designing TinyML models that perform well enough for a given application can be challenging. Instead of manually designing and testing models, it is possible to have a computer automatically generate and test models through a variety of techniques. These techniques all fall under the concept of AutoML [115].

AutoML techniques include methods for structuring data from raw formats, feature selection and extraction, model selection and hyperparameter optimisation.

One of the most researched parts of AutoML is Neural Architecture Search (NAS). A basic NAS searches for NN models to optimise a single metric – often accuracy [116]. However, especially in TinyML, researchers have proposed hardware aware NAS methods that optimise multiple metrics, e.g. both performance and resource metrics. A hardware Aware NAS typically returns a collection of models that form a Pareto frontier of models where each model is in some way better than every other model [117]. Therefore, Hardware Aware NASs can be used to generate near-optimal models for some given resource constraints.

NAS algorithms are often driven by a discrete optimisation algorithm such as an evolutionary algorithm or a reinforcement learning model [118]. State-of-the-art large-scale NAS algorithms are increasingly converting the discrete optimisation problem of selecting a neural architecture into a continuous optimisation problem to apply gradient-based optimisation of the architectures [119]. This trick speeds up

the NAS search, which can be extremely long for large-scale models.

9) COMPRESSION

Another way to reduce the memory consumption of a model is to apply a lossless compression algorithm to it, e.g. compressing a ML model using Huffman encoding.

While this will reduce the memory consumption of the model, it will also make the model slower to execute as it will first have to be extracted before being executed [12].

VI. DATA

At the fourth layer of the TinyML stack, we find the data that is fed into the rest of the TinyML system. We consider both the data used to train the systems and the data on which the system operates once it is operational.

A. DATASETS

TinyML applications often tackle unique challenges distinct from those of traditional scale machine learning. Therefore, many TinyML applications can not utilise the same datasets as traditional scale ML. This calls for specialized datasets for TinyML.

Unfortunately, the current landscape of TinyML datasets is scarcely populated, with only a handful of dedicated TinyML datasets available. For the majority of the time that the field has been active, it has been dominated by two datasets - Visual Wake Words [120] and Google Speech Commands [121]. Recently, researchers from Harvard University and the Technical University of Denmark launched a push to create high-quality TinyML datasets, starting the Wake Vision dataset [122].

Recent research suggests that TinyML models are less robust to label errors than traditional scale ML models [122]. This further motivates that the field of TinyML can benefit from specialized datasets that pay extra attention to minimizing the label error rate.

Visual Wake Words is a dataset containing images and associated labels of whether there is a person in an image. This dataset is useful for an application known as person detection, in which a TinyML system detects whether there is a person in an image. The dataset contains around 120.000 images derived from the COCO object detection dataset [120]. Studies estimate that the Visual Wake Words dataset has a label error rate of around 8% [122], severely limiting the performance of the TinyML systems that can be created using the dataset.

Google Speech Commands is a dataset containing audio samples along with labels of what word is spoken in an audio sample. The dataset is useful for keyword-spotting applications, which are often used to wake up more power-consuming systems, such as the digital assistants that most of us have on our smartphones. Google speech commands contain almost 100.000 audio samples and their associated keywords. Some keywords are classified as core words. These are the words that one can use the dataset to classify. Other keywords are considered auxiliary. Their function is to help a model distinguish core words from other words [121].

Wake Vision is, like Visual Wake Words, a dataset containing images and person labels. In fact, Wake Vision is designed to be a plug-in replacement for Visual Wake Words. Wake Vision is derived from the Open Images v7 dataset and is around 100 times larger than Visual Wake Words. In addition, the validation and test set of Wake Vision has been manually corrected to push the label error rate to 2.2%. With its increased size and lower label error rate, Wake Vision aims to enable much higher-quality TinyML research [122].

Three core datasets are not many for a field like TinyML, which aims to imbue billions of devices with diverse applications with intelligence. Furthermore, neither of the datasets focus on PdM. Therefore, we hope to see many more open datasets emerge in the coming years to assist TinyML in reaching many new applications. The Wake Vision paper proposes several ways to speed up the creation of TinyML datasets to aid in this goal [122].

Unfortunately for TinyML-based PdM, finding suitable data for predictive maintenance is notoriously challenging. One reason is that failures often lead to financial and reputational loss, especially in industry. Organizations, therefore, often go to great lengths to avoid failures – which is also the goal of PdM. Even when a failure occurs, the data about the failure is often not publicly released.

Although mostly toy datasets, we have identified three datasets that can be used for TinyML-based PdM until a better dataset is created.

ToyADMOS is a dataset which contains audio recordings of toys in normal and anomalous operating conditions [123]. The MLPerf Tiny benchmark, which, to our knowledge, is the only current PdM benchmark targeting TinyML, uses a subset of the ToyADMOS dataset [46].

MIMII is another dataset consisting of audio recordings. Unlike the ToyADMOS dataset, this dataset contains audio recordings of actual industrial machines in normal and anomalous operating conditions [124].

Turbofan Engine Degradation Dataset is a dataset of sensor readings of simulated turbofan engines as they degrade towards failure [125]. Each sensor reading contains information about the RUL of the turbofan engine and can therefore be used to train a RUL estimation model.

While these datasets provide a starting point for working with TinyML-based PdM, they have various flaws that prevent them from being used for high-quality TinyML-based PdM research. The ToyADMOS dataset is based on audio recordings of Toys, which can hardly be used for training production grade PdM models. The MIMII dataset, while consisting of audio recordings of actual industrial machinery,

TABLE 4. Overview of TinyML and PdM datasets.

Dataset	Size	For PdM
Visual Wake Words [120]	123,287 Images	✗
Google Speech Commands [121]	99,720 Audio Samples	✗
Wake Vision [122]	5,760,428 Images	✗
ToyADMOS [123]	>540 Hours of Audio	✓
MIMII [124]	32,157 Audio Samples	✓
Turbofan Engine Degradation Dataset [125]	1416 RUL Trajectories	✓

is limited to a few hours of normal sound and only around 15 minutes of anomalous sounds. The Turbofan Engine Degradation Dataset is specialized to a specific turbofan engine, which can not be used to train general PdM models.

A problem plaguing many PdM datasets is an imbalance of observations. By their nature, normal operating conditions are more frequent than anomalous; thus, most datasets include mostly normal observations. In Section V, we described methods to alleviate the imbalance problem. See Table 4 for a quick comparison of TinyML and PdM datasets.

B. INFERENCE DATA

Whereas TinyML models are almost always trained on larger computers, inference takes place in resource-constrained devices. As a result, the data captured and processed at inference time contributes to the total resource consumption of a TinyML system. Therefore, it can be beneficial to consider how data capturing and pre-processing can be optimized to balance performance and resource consumption at inference time. While much TinyML research has investigated ways to reduce the resource consumption of models and toolchains and increasing the resources provided by hardware, little attention has been paid to reducing the resources consumed by data processing and storage.

Research has shown that datasets are often collected and used without efficiency in mind [126], [127], leaving room for extensive optimisations. These optimisations can be done manually [127]; however, recent work on a Data Aware NAS proposes to search for optimal data granularities as a part of a hardware aware NAS. In doing so, the authors hypothesise that an optimal trade-off between resources dedicated to the model and data can be found. Data granularity refers to the concept that data can be captured and pre-processed in several ways to balance performance and resource consumption. For example, sound data can be collected at a reduced sample rate or lower bit width per sample [128].

VII. PREDICTIVE MAINTENANCE APPLICATIONS

At the top of the TinyML stack, we find the applications. With the focus of this review is PdM, we will primarily focus on PdM related applications. TinyML, however, sees applications in many other fields, including healthcare [16], environmental control [24], and the transportation sector [8].

As described in Section I, PdM systems are deployed on equipment to predict failures, so that maintenance can be conducted in advance, focusing on different objectives [2]. Below, we elaborate on some common ways to measure the performance of PdM systems [2].

Cost Minimization is a PdM objective that seeks to minimize the cost incurred by an organisation by the combination of failures and maintenance. Costs can include direct and indirect losses incurred by maintenance and failures, e.g., repair costs, lost production time or losses from unfulfilled orders.

Availability Maximization as an objective aims to maximize equipment uptime.

Safety is an objective that seeks to quantify and minimize the safety risks associated with equipment failure and maintenance.

Combination: Many PdM systems combine several objectives into a single goal for a PdM system. The individual objectives are often weighted to calculate a total performance score for the system.

Regardless of their objective, PdM finds applications in many fields. Below, we provide a generalised overview of application areas which are often referenced in the literature. These application areas are broad and encompass a wide array of subfields. While we can make general comments about these application areas, there will certainly be subfields for which these comments do not apply. For each application, we discuss the advantages of TinyML-based PdM compared to traditional cloud-based PdM. We also list publicly available datasets relevant to the application area known to the authors — if any. Finally, we discuss the typical resources available in the application area, e.g., energy, time, and cost.

Manufacturing: Manufacturing is arguably the model application area for PdM. This application area covers organizations whose main purpose is the production of goods. Much of the manufacturing relies heavily on costly equipment. This means that it is both expensive to replace failed equipment, and that equipment failures are likely to cause costly downtime. Therefore, the industry can benefit greatly from PdM efforts to reduce failures [129].

- **TinyML Advantages:** TinyML-based PdM allows for the use of PdM in industries where it has previously been infeasible to deploy such solutions. For example, an organization that produces medicine may be limited in what data they can transfer off-site due to regulations. TinyML further allows for plug-and-play PdM solutions that can be deployed by non-technical personnel and in hard-to-reach places.
- **Publicly Available Datasets:** Publicly available datasets that contain data relevant to PdM in Manufacturing include ToyADMOS [123] and MIMII [124].
- **Available Resources:** Most manufacturing settings have plenty of resources available. Resources can, however, be significantly limited in battery-powered scenarios.

Automotive: Contemporary vehicles capture an enormous amount of data about their operating condition and environment. This, coupled with vehicles being a safety-critical system, makes them an ideal candidate for PdM. Several research papers have been published about automotive PdM, ranging from tyre condition monitoring to faults in air pressure systems [130].

- **TinyML Advantages:** Vehicles often find themselves in areas with little or no connectivity. While traditional cloud-based PdM systems can not work in these conditions, it is not a problem for TinyML-based PdM systems. Furthermore, TinyML-based PdM systems can provide deterministic response times and increased reliability, which is essential in a safety-critical system.
- **Publicly Available Datasets:** Publicly available datasets that contain data relevant to PdM in the automotive industry is limited to ToyADMOS [123].
- **Available Resources:** The energy available in the automotive industry is primarily limited by the power generated by the engine for combustion vehicles or the battery for electric vehicles. Strict limits may be present on the response time of a PdM system.

Aerospace: Failures in the aerospace industry can have fatal consequences. Therefore, the industry is almost a perfect candidate for PdM [131].

- **TinyML Advantages:** TinyML-based PdM allow for PdM systems to be deployed without a reliable network connection, which is almost inevitable in aerospace systems. Furthermore, TinyML-based PdM systems can provide deterministic response times, which are critical to the real-time systems used in aerospace.
- **Publicly Available Datasets:** Publicly available datasets that contain data relevant to PdM in the aerospace industry is limited to the Turbofan Engine Degradation Dataset [125].
- **Available Resources:** The energy available in the aerospace industry is limited, and quick response times are essential. Costs are, however, less of an issue due to the already high equipment costs.

Healthcare: Another safety critical sector in which PdM has seen promising use is the healthcare industry. Medical devices today are some of the most complex systems in existence, and failures can bring catastrophic consequences for individuals [132].

- **TinyML Advantages:** Medical devices can benefit from TinyML-based PdM as they need to be deployed all over the world without assumptions on the available infrastructure. Wearables, in particular, can benefit from the low energy consumption of TinyML-based PdM. Furthermore, the privacy and security provided by TinyML-based PdM is desirable in many sensitive medical systems. Reliability and deterministic response times are further advantages that are important in medical devices.
- **Publicly Available Datasets:** Unfortunately, there are no publicly available datasets containing data relevant to PdM in the medical industry.
- **Available Resources:** The resources available to medical PdM systems are heavily dependent on the type of device and its location. That said, some medical applications require low energy, response time and cost.

Energy: The critically important energy sector sees heavy use of PdM systems both during production and transport [133].

- **TinyML Advantages:** TinyML-based PdM can find use in the energy sector as devices capable of functioning independently from the power grid that they are monitoring, e.g., by using batteries or energy harvesting techniques.
- **Publicly Available Datasets:** Publicly available datasets that contain data relevant to PdM in the energy industry is limited to MIMII [124].
- **Available Resources:** The energy resources available to PdM in the energy sector depend heavily on the source of the energy. Energy resources are not much of a concern for systems powered by off-grid generators or large batteries. Systems powered by smaller batteries or energy harvesting devices, however, need to minimize their energy use.

In all of the above application areas, PdM can be used to reduce the amount of critical failures that the equipment under monitoring will experience. By doing so, PdM reduces the number of equipment parts that must be replaced and, by extension, how many equipment parts must be manufactured. By reducing the number of parts which have to be manufactured, PdM contributes positively to several UN sustainability goals.

VIII. FUTURE RESEARCH DIRECTIONS

TinyML has received significant attention from the research community in recent years. Most research so far has focused on optimizing NN models to run in resource-constrained systems and for image classification applications. This leaves ample opportunity to contribute to the field of TinyML by focusing on other parts of the TinyML stack or on alternative applications. In this section, we shall discuss some future research directions that we find particularly promising.

Benchmarking TinyML Toolchains: This paper presented a comparison of TinyML toolchains in Section IV. This comparison did, however, not include an experimental evaluation of the resource requirements of the toolchains. For future work, a comprehensive experimental evaluation of the resource requirements and features of TinyML toolchains will be helpful for people getting started with TinyML. While most toolchains

provide estimates of their resource consumption and capabilities, there is a significant risk that these promises are biased.

Explainable AI for TinyML: A research topic currently receiving much attention in the conventional ML community is Explainable AI (XAI). XAI for TinyML systems has only received minor attention [134], and is therefore an interesting future research direction for TinyML.

Datasets for TinyML-based PdM: As highlighted in Section VI-A, the field of TinyML currently lacks many high-quality datasets. This is especially the case for PdM, where the current standard dataset is based on audio recordings of toys [123]. It would be interesting for new PdM datasets to incorporate multimodal data to allow for high-quality research into multimodal TinyML models.

Energy Harvesting and Intermittent Learning: As highlighted in Section III-E,battery-driven TinyML devices are significantly less sustainable than their directly connected or energy-harvesting counterparts. As it is often not possible to connect TinyML devices directly to the power grid, more research into energy harvesting for TinyML is warranted. Energy harvesting systems can not often guarantee a reliable source of power. Creating TinyML systems which work intermittently when power is available should, therefore, be a focus of future research.

New Computing Paradigms: Multiple new computing paradigms are currently being investigated to improve the performance and efficiency of TinyML systems. We discussed many of these in Section III-A. Further research into these new computing paradigms could catapult the TinyML field to new heights.

Advancing TinyML NAS: NAS has already proven itself a valuable tool for creating TinyML systems [66], [128], [135]. Because of the manual labour required for actual deployment, current NAS systems often use unreliable proxies for hardware-specific metrics such as inference time, memory consumption and energy consumption. Creating reliable tools to estimate these hardware-specific metrics without deployment could significantly improve the performance of NAS for TinyML.

Standardisation Efforts: One of TinyML's biggest hurdles is going from being used in research and hobby projects to being used in large-scale professional deployments. Standardisation efforts could contribute to this transition by streamlining the deployment of TinyML solutions. Such standardisation efforts are already ongoing in other parts of the PdM field [136], [137]. As with all standardisation efforts, a balance must be found between giving time for innovation and the benefits brought on by standardisation. Some parts of the TinyML stack have reached higher maturity levels than others and are, in our opinion, ready to be considered for standardisation efforts. In particular, TinyML models and their optimisation have been heavily researched, and thus, standardisation efforts into model formats (e.g., ONNX) and optimisation methods (e.g., quantisation and pruning) could prove beneficial.

IX. CONCLUSION

TinyML-based PdM is a promising technology that can enable many new PdM applications. This review has been dedicated to presenting a holistic view of the field that can help new practitioners or researchers rapidly get up to speed with the latest developments in the field. Unlike many other reviews/surveys of the field, this paper encompasses the entirety of the TinyML stack from hardware to data and PdM applications. We also discuss the sustainability of TinyML-based PdM, particularly the impact of TinyML hardware and the sustainability of PdM applications.

While this paper offers a comprehensive overview of TinyML and PdM, it is important to acknowledge certain limitations that suggest areas for further exploration. Specifically, this paper has not discussed the deployment and update of TinyML systems. Additionally, several key topics closely related to TinyML—such as the Internet of Things (IoT), Federated Learning, and Distributed Computing—are not covered in detail. Finally, although a variety of tools and toolchains were reviewed, the depth of available options and their feature sets exceeded the scope of this work, leaving room for further investigation of those tools in future studies.

REFERENCES

[1] A. T. Tunggal. *The Cost of Downtime at the World's Biggest Online Retailer*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer

[2] T. Zhu, Y. Ran, X. Zhou, Y. Wen, and R. Deng, "A survey of predictive maintenance: Systems, purposes and approaches," 2019, *arXiv:1912.07383*.

[3] M. Bevilacqua and M. Braglia, "The analytic hierarchy process applied to maintenance strategy selection," *Rel. Eng. Syst. Saf.*, vol. 70, no. 1, pp. 71–83, Oct. 2000.

[4] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, Jul. 2009.

[5] A. M. Hussein, A. K. Idrees, and R. Couturier, "Distributed energy-efficient data reduction approach based on prediction and compression to reduce data transmission in IoT networks," *Int. J. Commun. Syst.*, vol. 35, no. 15, p. 5282, Oct. 2022.

[6] TinyML Found. *About us TinyML*. Accessed: Sep. 17, 2024. [Online]. Available: https://www.tinyml.org/

[7] E. Njor, J. Madsen, and X. Fafoutis, "A primer for tinyML predictive maintenance: Input and model optimisation," in *Proc. IFIP Int. Conf. Artif. Intell. Appl. Innov.*. Cham, Switzerland: Springer, Jan. 2022, pp. 67–78.

[8] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-enabled frugal smart objects: Challenges and opportunities," *IEEE Circuits Syst. Mag.*, vol. 20, no. 3, pp. 4–18, 3rd Quart., 2020.

[9] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[10] Y. Y. Siang, M. R. Ahamd, and M. S. Z. Abidin, "Anomaly detection based on tiny machine learning: A review," *Open Int. J. Informat.*, vol. 9, no. 2, pp. 67–78, 2021.

[11] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: Current progress, research challenges, and future roadmap," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 1303–1306.

[12] D. L. Dutta and S. Bharali, "TinyML meets IoT: A comprehensive survey," *Internet Things*, vol. 16, Dec. 2021, Art. no. 100461.

[13] H. Yelchuri and R. Rashmi, "A review of TinyML," 2022, *arXiv:2211.04448*.

[14] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, Apr. 2022.

[15] S. O. Ooko, M. M. Ogore, J. Nsenga, and M. Zennaro, "TinyML in Africa: Opportunities and challenges," in *Proc. IEEE GLOBECOM Workshops (GC Wkshps)*, Dec. 2021, pp. 1–6.

[16] V. Tsoukas, E. Boumpa, G. Giannakas, and A. Kakarountas, "A review of machine learning and TinyML in healthcare," in *Proc. 25th Pan-Hellenic Conf. Informat.*, Nov. 2021, pp. 69–73.

[17] H. Han and J. Siebert, "TinyML: A systematic review and synthesis of existing research," in *Proc. Int. Conf. Artif. Intell. Inf. Commun. (ICAIIC)*, Feb. 2022, pp. 269–274.

[18] H. Bamoumen, A. Temouden, N. Benamar, and Y. Chtouki, "How TinyML can be leveraged to solve environmental problems: A survey," in *Proc. Int. Conf. Innov. Intell. Informat., Comput., Technol. (3ICT)*, Nov. 2022, pp. 338–343.

[19] R. Immonen and T. Hämäläinen, "Tiny machine learning for resource-constrained microcontrollers," *J. Sensors*, vol. 2022, Nov. 2022, Art. no. 7437023.

[20] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," *IEEE Sensors J.*, vol. 22, no. 22, pp. 21362–21390, Nov. 2022.

[21] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid, "A comprehensive survey on TinyML," *IEEE Access*, vol. 11, pp. 96892–96922, 2023.

[22] V. Rajapakse, I. Karunanayake, and N. Ahmed, "Intelligence at the extreme edge: A survey on reformable TinyML," *ACM Comput. Surv.*, vol. 55, no. 13s, pp. 1–30, Dec. 2023.

[23] S. B. Lakshman and N. U. Eisty, "Software engineering approaches for TinyML based IoT embedded vision: A systematic literature review," in *Proc. IEEE/ACM 4th Int. Workshop Softw. Eng. Res. Practices IoT (SERP4IoT)*, May 2022, pp. 33–40.

[24] Istofa, P. Prajitno, and I. P. Susila, "A systematic literature review of TinyML for environmental radiation monitoring system," in *Proc. 6th Mech. Eng., Sci. Technol. Int. Conf. (MEST)*. Amsterdam, The Netherlands: Atlantis Press, 2023, pp. 461–473.

[25] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, and S. Han, "Tiny machine learning: Progress and futures [feature]," *IEEE Circuits Syst. Mag.*, vol. 23, no. 3, pp. 8–34, Mar. 2023.

[26] M. Bhamare, P. V. Kulkarni, R. Rane, S. Bobde, and R. Patankar, "Tinyml applications and use cases for healthcare," in *TinyML for Edge Intelligence in IoT and LPWAN Networks*. Amsterdam, The Netherlands: Elsevier, 2024, pp. 331–353.

[27] B. S. Chaudhari, S. N. Ghorpade, M. Zennaro, and R. Paškauskas, "TinyML for low-power Internet of Things," in *TinyML for Edge Intelligence in IoT and LPWAN Networks*. Amsterdam, The Netherlands: Elsevier, 2024, pp. 1–12.

[28] V. Tsoukas, A. Gkogkidis, E. Boumpa, and A. Kakarountas, "A review on the emerging technology of TinyML," *ACM Comput. Surv.*, vol. 56, no. 10, pp. 1–37, Oct. 2024.

[29] L. Capogrosso, F. Cunico, D. S. Cheng, F. Fummi, and M. Cristani, "A machine learning-oriented survey on tiny machine learning," *IEEE Access*, vol. 12, pp. 23406–23426, 2024.

[30] H. Oufettoul, R. Chaibi, and S. Motahhir, "TinyML applications, research challenges, and future research directions," in *Proc. 21st Learn. Technol. Conf. (L&T)*, Jan. 2024, pp. 86–91.

[31] P. Warden and D. Situnayake, *TinyML: Machine Learning With TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. Sebastopol, CA, USA: O'Reilly Media, 2019.

[32] G. M. Iodice, *TinyML Cookbook: Combine Artificial Intelligence and Ultra-Low-Power Embedded Devices to Make the World Smarter*. Birmingham, U.K.: Packt Publishing, 2022.

[33] H. E. C. Lab, *Machine Learning Systems*. San Francisco, CA, USA: GitHub, Harvard Edge Computing Lab, 2024. [Online]. Available: https://harvard-edge.github.io/cs249r_book/

[34] ARM. *Processor IP for the Widest Range of Devices*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu

[35] Espressif. *Esp32*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.espressif.com/en/products/socs/esp32

[36] A. Garofalo, Y. Tortorella, M. Perotti, L. Valente, A. Nadalini, L. Benini, D. Rossi, and F. Conti, "DARKSIDE: A heterogeneous RISC-V compute cluster for extreme-edge on-chip DNN inference and training," *IEEE Open J. Solid-State Circuits Soc.*, vol. 2, pp. 231–243, 2022.

[37] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[38] ARM. *Ethos—NPUs*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu?families=ethos

[39] Google. *Coral*. Accessed: Dec. 7, 2024. [Online]. Available: https://coral.ai/

[40] Renesas. *The Cost of Downtime at the World's Biggest Online Retailer*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.renesas.com/us/en/software-tool/ai-accelerator-drp-ai?gad_source=1

[41] Nvidia. *Nvidia Jetson Nano*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/

[42] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnol.*, vol. 15, no. 7, pp. 529–544, Jul. 2020.

[43] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," 2017, *arXiv:1705.06963*.

[44] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proc. IEEE*, vol. 108, no. 12, pp. 2108–2135, Dec. 2020.

[45] W. Haensch, T. Gokmen, and R. Puri, "The next generation of deep learning hardware: Analog computing," *Proc. IEEE*, vol. 107, no. 1, pp. 108–122, Jan. 2019.

[46] C. Banbury, V. J. Reddi, P. Torelli, N. Jeffries, C. Kiraly, J. Holleman, P. Montino, D. Kanter, P. Warden, D. Pau, U. Thakker, A. Torrini, J. Cordaro, G. Di Guglielmo, J. Duarte, H. Tran, N. Tran, N. Wenxu, and X. Xuesong, "MLPerf tiny benchmark," in *Proc. Neural Inf. Process. Syst. Track Datasets Benchmarks*, vol. 1, J. Vanschoren and S. Yeung, Eds., 2021, pp. 1–11. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/da4fb5c6e93e74d3df8527599fa62642-Paper-round1.pdf

[47] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J.-S. Seo, J. Sieracki, U. Thakker, M. Verhelst, and P. Yadav, "Benchmarking TinyML systems: Challenges and direction," 2020, *arXiv:2003.04821*.

[48] P. Bombelli, A. Savanth, A. Scarampi, S. J. L. Rowden, D. H. Green, A. Erbe, E. Årstøl, I. Jevremovic, M. F. Hohmann-Marriott, S. P. Trasatti, E. Ozer, and C. J. Howe, "Powering a microprocessor by photosynthesis," *Energy Environ. Sci.*, vol. 15, no. 6, pp. 2529–2536, 2022.

[49] S. Prakash, M. Stewart, C. Banbury, M. Mazumder, P. Warden, B. Plancher, and V. J. Reddi, "Is TinyML sustainable?" *Commun. ACM*, vol. 66, no. 11, pp. 68–77, Nov. 2023.

[50] Arduino. *Nano 33 Ble Sense*. Accessed: Dec. 7, 2024. [Online]. Available: https://docs.arduino.cc/hardware/nano-33-ble-sense

[51] *SparkFun Edge Development Board—Apollo3 Blue*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.sparkfun.com/products/15170

[52] Espressif. *Esp32-Devkitc*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.espressif.com/en/products/devkits/esp32-devkitc/overview

[53] G. Bekaroo and A. Santokhee, "Power consumption of the raspberry Pi: A comparative analysis," in *Proc. IEEE Int. Conf. Emerg. Technol. Innov. Bus. Practices Transformation Societies (EmergiTech)*, Aug. 2016, pp. 361–366.

[54] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes, "TensorFlow lite micro: Embedded machine learning for tinyml systems," in *Proc. Mach. Learn. Syst.*, vol. 3, 2021, pp. 800–811.

[55] EdgeImpulse. *Edge Impulse—The Leading Edge AI Platform*. Accessed: Dec. 7, 2024. [Online]. Available: https://edgeimpulse.com
[56] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm Cortex-M CPUs," 2018, *arXiv:1801.06601*.
[57] STMicroelectronics. *STM32cube.AI—Stmicrolectronics—STM32 AI*. Accessed: Dec. 7, 2024. [Online]. Available: https://stm32ai.st.com/stm32-cube-ai/
[58] STMicroelectronics. *NanoEdgeAIStudio—Automated Machine Learning (ML) Tool for STM32 Developers—Stmicroelectronics*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.st.com/en/development-tools/nanoedgeaistudio.html
[59] Ekkono. *Edge Machine Learning for IoT*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.ekkono.ai
[60] Renesas. *E-AI Development Environment for Microcontrollers*. Accessed: Dec. 7, 2024. [Online]. Available: https://www.renesas.com/us/en/e-ai-development-environment-microcontrollers
[61] UTensor. *Utensor/Utensor: TinyML AI Inference Library*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/uTensor/uTensor
[62] eloquentarduino. *TinyML Gen*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/eloquentarduino/tinymlgen
[63] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 871–875, May 2020.
[64] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "FANN-on-MCU: An open-source toolkit for energy-efficient neural network inference at the edge of the Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4403–4417, May 2020.
[65] Apache. *MicroTVM Design Document*. Accessed: Feb. 8, 2023. [Online]. Available: https://tvm.apache.org/docs/arch/microtvm_design.html#microtvm-design
[66] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 11711–11722.
[67] C. B. Moretti. *Moretticb/Neurona: Artificial Neural Networks for Arduino*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/moretticb/Neurona
[68] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.
[69] eloquentarduino. *Introducing MicroML*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/eloquentarduino/micromlgen
[70] D. Morawiec. *Sklearn-Porter*. transpile trained scikit-learn estimators to C, Java, JavaScript others. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/nok/sklearn-porter
[71] D. Morawiec. *Sklearn-Porter*. weka-porter. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/nok/weka-porter
[72] L. T. da Silva, V. M. A. Souza, and G. E. A. P. A. Batista, "EmbML tool: Supporting the use of supervised learning algorithms in low-cost embedded systems," in *Proc. IEEE 31st Int. Conf. Tools Artif. Intell. (ICTAI)*, Nov. 2019, pp. 1633–1637.
[73] Renesas. *Reality AI Software for Real Time Analytics on MCUs & MPUs*. Accessed: Jun. 13, 2024. [Online]. Available: https://www.renesas.com/us/en/products/microcontrollers-microprocessors/reality-ai
[74] J. Nordby, "Emlearn: Machine learning inference engine for microcontrollers and embedded devices," Zendo, CERN, Mar. 2019, doi: 10.5281/zenodo.2589394.
[75] TensorFlow. *TensorFlow Lite for Microcontrollers*. Accessed: Feb. 4, 2024. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers
[76] Microsoft. *Embedded Learning Library (Ell)*. Accessed: Dec. 7, 2024. [Online]. Available: https://microsoft.github.io/ELL/
[77] ARM. *Arm-Software/Armnn*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/ARM-software/armnn
[78] ARM. *Arm-Software/CMSIS-NN*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/ARM-software/CMSIS-NN
[79] Fraunhofer-IMS. *Frauhofer-IMS/AlfES_for_Arduino*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/Fraunhofer-IMS/AIfES_for_Arduino
[80] EEESlab. *EEESlab/CMix-NN*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/EEESlab/CMix-NN
[81] pulp platform. *Pulp-Platform/FANN-on-MCU*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/pulp-platform/fann-on-mcu
[82] Mit-Han Lab. *Mit-Han-Lab/Tinyengine*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/mit-han-lab/tinyengine
[83] BayesWitnesses. *Bayeswitnesses/M2Cgen*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/BayesWitnesses/m2cgen
[84] lucastsutsui. *Lucastsutsui/EmbML*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/lucastsutsui/EmbML
[85] emlearn. *Emlearn/Emlearn*. Accessed: Dec. 7, 2024. [Online]. Available: https://github.com/emlearn/emlearn
[86] H. Ren, D. Anicic, and T. A. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–8.
[87] C. Profentzas, M. Almgren, and O. Landsiedel, "MicroTL: Transfer learning on low-power IoT devices," in *Proc. IEEE 47th Conf. Local Comput. Netw. (LCN)*, Sep. 2022, pp. 1–8.
[88] H. Cai, C. Gan, L. Zhu, and S. Han, "TinyTL: Reduce memory, not parameters for efficient on-device learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 11285–11297.
[89] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256 KB memory," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 22941–22954.
[90] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–12.
[91] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and <0.5MB model size," 2016, *arXiv:1602.07360*.
[92] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[93] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.
[94] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2820–2828.
[95] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. Int. Conf. Mach. Learn.*, Jan. 2019, pp. 6105–6114.
[96] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once for all: Train one network and specialize it for efficient deployment," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–15. [Online]. Available: https://arxiv.org/pdf/1908.09791.pdf
[97] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–9. [Online]. Available: https://arxiv.org/pdf/1812.00332.pdf
[98] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for MobileNetV3," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 1314–1324.
[99] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Memory-efficient patch-based inference for tiny deep learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 2346–2358.
[100] F. K.-D. Noering, Y. Schroeder, K. Jonas, and F. Klawonn, "Pattern discovery in time series using autoencoder in comparison to nonlearning approaches," *Integr. Comput.-Aided Eng.*, vol. 28, no. 3, pp. 237–256, Jun. 2021.
[101] L. Bernstein, A. Sludds, R. Hamerly, V. Sze, J. Emer, and D. Englund, "Freely scalable and reconfigurable optical hardware for deep learning," *Sci. Rep.*, vol. 11, no. 1, pp. 1–12, Feb. 2021.
[102] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," 2021, *arXiv:2106.08295*.
[103] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[104] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Enabling AI at the edge with XNOR-networks," *Commun. ACM*, vol. 63, no. 12, pp. 83–90, Nov. 2020.

[105] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag, "What is the state of neural network pruning?" *Proc. Mach. Learn. Syst.*, vol. 2, pp. 129–146, Mar. 2020.

[106] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, 2017, pp. 548–560.

[107] "Nvidia a100 tensor core GPU architecture," NVIDIA, Santa Clara, CA, USA, Tech. Rep. 2020. Accessed: Dec. 7, 2024. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[108] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*.

[109] G. Zhong, W. Liu, H. Yao, T. Li, J. Sun, and X. Liu, "Merging similar neurons for deep networks compression," *Cogn. Comput.*, vol. 12, no. 3, pp. 577–588, May 2020.

[110] X. Liu, W. Liu, L. Wang, and G. Zhong, "Deep architecture compression with automatic clustering of similar neurons," in *Proc. 4th Chin. Conf. Pattern Recognit. Comput. Vis. (PRCV)*, Beijing, China. Cham, Switzerland: Springer, Nov. 2021, pp. 361–373.

[111] R. Harang and H. Sanders, "Magnificent minified models," 2023, *arXiv:2306.10177*.

[112] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*.

[113] P. Zalewski, L. Marchegiani, A. Elsts, R. Piechocki, I. Craddock, and X. Fafoutis, "From bits of data to bits of knowledge—An on-board classification framework for wearable sensing systems," *Sensors*, vol. 20, no. 6, p. 1655, Mar. 2020.

[114] N. P. Ghanathe and S. Wilton, "T-RecX: Tiny-resource efficient convolutional neural networks with early-eXit," in *Proc. 20th ACM Int. Conf. Comput. Frontiers*, May 2023, pp. 123–133.

[115] A. Zender and B. G. Humm, "Ontology-based meta AutoML," *Integr. Comput.-Aided Eng.*, vol. 29, no. 4, pp. 351–366, Aug. 2022.

[116] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 1997–2017, 2019.

[117] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via Lamarckian evolution," in *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*, New Orleans, LA, USA, May 2019, pp. 1–16. [Online]. Available: https://openreview.net/forum?id=ByME42AqK7

[118] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, "A comprehensive survey on hardware-aware neural architecture search," 2021, *arXiv:2101.09336*.

[119] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–13. [Online]. Available: https://openreview.net/forum?id=S1eYHoC5FX

[120] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes, "Visual wake words dataset," 2019, *arXiv:1906.05721*.

[121] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.

[122] C. Banbury, E. Njor, A. M. Garavagno, M. Stewart, P. Warden, M. Kudlur, N. Jeffries, X. Fafoutis, and V. J. Reddi, "Wake vision: A tailored dataset and benchmark suite for TinyML computer vision applications," 2024, *arXiv:2405.00892*.

[123] Y. Koizumi, S. Saito, H. Uematsu, N. Harada, and K. Imoto, "ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection," in *Proc. IEEE Workshop Appl. Signal Process. Audio Acoust. (WASPAA)*, Oct. 2019, pp. 313–317.

[124] H. Purohit, R. Tanabe, T. Ichige, T. Endo, Y. Nikaido, K. Suefusa, and Y. Kawaguchi, "MIMII dataset: Sound dataset for malfunctioning industrial machine investigation and inspection," in *Proc. Detection Classification Acoustic Scenes Events Workshop (DCASE)*, 2019, pp. 1–20.

[125] A. Saxena and K. Goebel, "Turbofan engine degradation simulation data set," NASA Ames Prognostics Data Repository, 2008, vol. 18, pp. 878–887.

[126] A. Khan, N. Hammerla, S. Mellor, and T. Plötz, "Optimising sampling rates for accelerometer-based human activity recognition," *Pattern Recognit. Lett.*, vol. 73, pp. 33–40, Apr. 2016.

[127] X. Fafoutis, L. Marchegiani, A. Elsts, J. Pope, R. Piechocki, and I. Craddock, "Extending the battery lifetime of wearable sensors with embedded machine learning," in *Proc. IEEE 4th World Forum Internet Things (WF-IoT)*, Feb. 2018, pp. 269–274.

[128] E. Njor, J. Madsen, and X. Fafoutis, "Data aware neural architecture search," 2023, *arXiv:2304.01821*.

[129] Z. M. Çınar, A. A. Nuhu, Q. Zeeshan, O. Korhan, M. Asmael, and B. Safaei, "Machine learning in predictive maintenance towards sustainable smart manufacturing in Industry 4.0," *Sustainability*, vol. 12, no. 19, p. 8211, Oct. 2020.

[130] A. Theissler, J. Pérez-Velázquez, M. Kettelgerdes, and G. Elger, "Predictive maintenance enabled by machine learning: Use cases and challenges in the automotive industry," *Rel. Eng. Syst. Saf.*, vol. 215, Nov. 2021, Art. no. 107864.

[131] B. Shukla, I.-S. Fan, and I. Jennions, "Opportunities for explainable artificial intelligence in aerospace predictive maintenance," in *Proc. PHM Soc. Eur. Conf.*, Jul. 2020, vol. 5, no. 1, p. 11.

[132] O. Manchadi, F.-E. Ben-Bouazza, and B. Jioudi, "Predictive maintenance in healthcare system: A survey," *IEEE Access*, vol. 11, pp. 61313–61330, 2023.

[133] E. Jovicic, D. Primorac, M. Cupic, and A. Jovic, "Publicly available datasets for predictive maintenance in the energy sector: A review," *IEEE Access*, vol. 11, pp. 73505–73520, 2023.

[134] M. Sabih, M. Yayla, F. Hannig, J. Teich, and J.-J. Chen, "Robust and tiny binary neural networks using gradient-based explainability methods," in *Proc. 3rd Workshop Mach. Learn. Syst.*, May 2023, pp. 87–93.

[135] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. J. Reddi, M. Mattina, and P. Whatmough, "MicroNets: Neural network architectures for deploying TinyML applications on commodity microcontrollers," in *Proc. Mach. Learn. Syst.*, vol. 3, 2020, pp. 517–532.

[136] M. Weise, M. Böhms, D. Allaix, A. Sánchez-Rodríguez, and M. Rigotti, "Importance of digitalization and standardization for bridge and tunnel monitoring and predictive maintenance," *Ce/Papers*, vol. 6, no. 5, pp. 592–599, Sep. 2023.

[137] K. Kaur, M. Selway, G. Grossmann, M. Stumptner, and A. Johnston, "Towards an open-standards based framework for achieving condition-based predictive maintenance," in *Proc. 8th Int. Conf. Internet Things*, Oct. 2018, pp. 1–8.

EMIL NJOR received the B.S. and M.S. degrees in computer science from Aalborg University, Aalborg, Denmark, in 2019 and 2021, respectively. He is currently pursuing the Ph.D. degree in computer science with the Technical University of Denmark, Kongens Lyngby, Denmark.

MOHAMMAD AMIN HASANPOUR (Graduate Student Member, IEEE) received the B.Sc. degree in electrical engineering from the K. N. Toosi University of Technology, Tehran, Iran, in 2019, and the M.Sc. degree in electrical engineering from the Sharif University of Technology, Tehran, in 2021. He is currently pursuing the Ph.D. degree in computer science with the Technical University of Denmark (DTU), Kongens Lyngby, Denmark.

JAN MADSEN (Senior Member, IEEE) received the Ph.D. degree in computer science from the Technical University of Denmark, Kongens Lyngby, Denmark. He is currently a Full Professor in computer-based systems and the Head of the Department of Applied Mathematics and Computer Science (DTU Compute), Technical University of Denmark. His research interests include the intersection between computer science and biotechnology, with a special focus on design, modeling, construction of microelectronic (MPSoC and IoT), microfluidic (lab-on-chip), and microbiological (molecular) computing systems, including the development of design automation tools and design methodologies. He is a Board Member of EDAA and a fellow of the Danish Academy of Technical Sciences, where he is a member of the Council of Digital Experts and serves in the Council for Technology and Society. He is a member of ACM and a fellow of DATE.

XENOFON FAFOUTIS (Senior Member, IEEE) received the B.Sc. degree in informatics and telecommunications from the University of Athens, Greece, in 2007, the M.Sc. degree in computer science from the University of Crete, Greece, in 2010, and the Ph.D. degree in embedded systems engineering from the Technical University of Denmark, in 2014. He is currently a Professor with the Embedded Systems Engineering (ESE) Section, Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU Compute). His research interests include wireless embedded systems as an enabling technology for digital health, smart cities, and the (Industrial) Internet of Things (IoT).