Hindawi
Journal of Sensors
Volume 2022, Article ID 7437023, 11 pages
https://doi.org/10.1155/2022/7437023

Review Article

Tiny Machine Learning for Resource-Constrained Microcontrollers

Riku Immonen and Timo Hämäläinen

Faculty of Information Technology, University of Jyväskylä, P.O. Box 35, FI-40014, Finland

Correspondence should be addressed to Riku Immonen; riku.j.immonen@jyu.fi

Received 27 April 2022; Revised 18 September 2022; Accepted 3 November 2022; Published 10 November 2022

Academic Editor: Jiacheng Yang

We use 250 billion microcontrollers daily in electronic devices that are capable of running machine learning models inside them. Unfortunately, most of these microcontrollers are highly constrained in terms of computational resources, such as memory usage or clock speed. These are exactly the same resources that play a key role in teaching and running a machine learning model with a basic computer. However, in a microcontroller environment, constrained resources make a critical difference. Therefore, a new paradigm known as tiny machine learning had to be created to meet the constrained requirements of the embedded devices. In this review, we discuss the resource optimization challenges of tiny machine learning and different methods, such as quantization, pruning, and clustering, that can be used to overcome these resource difficulties. Furthermore, we summarize the present state of tiny machine learning frameworks, libraries, development environments, and tools. The benchmarking of tiny machine learning devices is another thing to be concerned about; these same constraints of the microcontrollers and diversity of hardware and software turn to benchmark challenges that must be resolved before it is possible to measure performance differences reliably between embedded devices. We also discuss emerging techniques and approaches to boost and expand the tiny machine learning process and improve data privacy and security. In the end, we form a conclusion about tiny machine learning and its future development.

1. Introduction

Globally, Internet of Things (IoT) devices are sending data to the cloud at an accelerating rate because the number of such devices is increasing, and the capacity of network connections is improving all the time. At the same time, the number of different cloud service platforms has grown, and these platforms have become more accessible for all users. International Data Corporation estimates that 79.4 zettabytes of data will be generated by 41.6 billion IoT devices in 2025 [1]. However, it is not necessary to transfer all these data to the cloud when we can take advantage of the edge computing capabilities of the IoT devices instead of using cloud computing, which burdens networks and radio bands. Furthermore, applications may also need to be geographically dispersed, higher bandwidth, ultralow latency, and privacy-sensitive features [2]. Hence, a computing paradigm that happens closer to the edge is needed. Therefore, the use of IoT devices as comprehensive edge computers is an important issue to address. The authors of [2] explained a taxonomy of different computing paradigms such as fog, edge, extreme edge, and mist computing that can be found in the literature. In this review, for clarity, we use cloud–fog–edge taxonomy, with the common term ‘edge’ to mean edge computing that happens in the cloud or fog-connected sensor nodes or IoT devices themselves. Edge computing—or, more specifically, Edge artificial intelligence (Edge AI)—means the computation of a machine learning (ML) algorithm on an edge device or node [3] that can be as tiny as a single microcontroller with integrated IoT radio. Today, the term TinyML [4] is widely used in the context of a lightweight ML for embedded devices. In addition, these edge devices are now located more often at the edge of the physical world, measuring some physical quantity. When defining TinyML hardware or devices, ultralow power consumption is the most defining characteristic, typically below 1 mW [5]. In this way, the processor range includes 32-bit Arm Cortex-M7 and RISC-V PULP processors and below. TinyML is a vast research topic, and its main elements can be divided roughly into datasets, use cases, hardware, framework and algorithms, and models [6]. In the end, all these main elements focus more or less on the technological improvement of products.

Edge AI is also changing sensors when sensor manufacturers have recently started implementing AI features in sensors' application specific integrated circuits (ASIC) such as STMicroelectronics' Intelligent Sensor Processing Units (ISPU) [7]. In addition, STMicroelectronics introduced a new generation of microelectromechanical system (MEMS) sensors in 2022, including ISPUs and support for on-device learning. Another self-learning AI sensor manufacturer is Bosch Sensortec, which promises always-on data processing algorithms at ultralow power consumption in their new MEMS-integrated 32-bit Fuser2 microcontrollers [8]. TinyML is a promising technology for soft sensors and sensor data fusion, and it has been recently studied in articles [9, 10, 11].

2. Challenges and Opportunities

TinyML has big challenges, such as the microcontroller's limited computational resources, lack of a unified framework [12], and absence of open-source TinyML datasets [5]. TinyML also has huge potential because microcontrollers are cheap and widespread. TinyML can potentially decrease the IoT device's energy consumption, lifetime costs, and inference latency. Same time it increases IoT device's data privacy and intelligence level. Table 1 summarizes the advantages and disadvantages of application features in IoT devices.

Table 1: TinyML application features in IoT devices.

Feature	Increased	Decreased
Energy usage	CPU	Radio, overall
Processor	Throughput	Latency
Memory	Allocation complexity	Footprint
Data	Security, privacy	Transfer need
ML model	Optimization demand	Size, accuracy
Costs	Software dev.	Hardware

Edge computing requires a great deal of resources from an IoT device, as the computational resources of its microcontroller are typically limited. This computational challenge becomes overwhelming if the standard neural networks- (NN-) based ML algorithms are used. For this reason, light software frameworks, tools, and libraries have recently been created, especially for use with microcontrollers, that can be used to build the TinyML model. After the model is created, it can be used within the source code to implement ML features in IoT devices. Nevertheless, one of the great challenges in TinyML is the lack of a unified framework that can be used across a wide range of hardware [12]. The fundamental difference between normal ML and TinyML procedures is that with the latter, the model is usually created on a more efficient computer and then ported to a microcontroller, which begins to perform inference based on the model [13–15]. The edge inference throughput should be robust without frame drops [16]. Another thing is, that the microcontroller running the ML algorithm in a loop is not the only process that consumes energy and computational resources on an edge IoT device. Raw data measurement and communication with the cloud services and the other IoT devices are also resource-expensive processes that the edge device must typically perform. Resource optimization becomes even more important to IoT devices that are battery-powered. Therefore, computational resource optimization is a rising topic that has been addressed in recent TinyML and edge computing articles using different techniques. Another challenge is the absence of TinyML-focused open-source datasets that are large enough for TinyML benchmarking and academic research [5]. Additionally, it would be good if the data in the datasets corresponded to the data sent by external sensors in terms of temporal and spatial resolution [17].

TinyML has enormous potential because there are 250 billion microcontrollers in our printers, TVs, cars, and pacemakers that are capable of running the ML model on the edge [18]. It is also estimated that 2.5 billion IoT devices will be shipped with a TinyML chipset in 2030 [19]. These huge estimations lean on the fact that microcontrollers are astonishingly cheap, and that the world has only just started its digital transformation and will need much more data in the future. For this reason, TinyML has been defined as a fast-growing field among machine learning technologies [4]. In addition, cloud-sourced ML inference could cost a lot during an AI application's lifetime because some applications are very data-centered. Therefore, ML inference and data pruning are much cheaper to perform at the data origin [3]. This approach also saves energy because the use of IoT radio is a more energy-expensive operation than edge computing. Moreover, when ML inference is made in an IoT device itself, it reduces latency and increases data privacy and security [20].

3. Methods for Resource Optimization

Different approaches and methods can be used to save microcontrollers' computational resources, such as memory and processor usage, when used in TinyML devices. One of the most commonly used methods to reduce the computational load on basic edge devices is simply to send heavy computational tasks forward to edge gateways, as was done by the authors in [21]. Nevertheless, this approach could lead to rising power consumption in TinyML devices because sending and receiving data is a more energy-consuming process than powering an on-device neural network (NN) [22], and this can be critical, at least for battery-operated devices. Therefore, it is better to resolve a resource problem by computational means within the edge device itself. This can be done through quantization, pruning, and clustering methods that reduce ML model size and processor usage. The downside with the use of these tradeoff methods is that the prediction or classification accuracy usually decreases.

3.1. Quantization. One of the vital computational capabilities of a microcontroller, which is usually needed for running NNs, is performing floating-point operations. Neural networks typically use high-precision 32-bit floating-point data in the production and inference mode [23]. However, these floating-point NN operations require a great deal of memory, system throughput, and clock speed from a microcontroller [24]. In some cases, microcontrollers are not even capable of performing hardware floating-point operations; e.g., in the Arm Cortex-M processor series, the hardware floating-point unit (FPU) is included, starting with M4F processors [25]. Still, this problem can be solved by computational means by using the Arm software floating-point C library, software floating-point emulation (FPE), or converting floating-point data to fixed-point data format [26, 27]. Quantization of 32-bit floating-point data to 8-bit fixed-point data lowers the model's memory footprint by 75%, and integer operations make the microcontroller run much faster [28]. In [29], the authors tested fine-tuned convolutional neural network (CNN) quantization with the CIFAR-10 dataset, 30 epochs, and different weight and activation bit-width combinations. The results showed that by using 4-bit fixed-point weight and activation values, the classification error rate gained only from 6.98% to 8.30%, compared to floating-point values. The authors of [30] also reported good results when testing 4-bit precision quantization with different datasets; they reported 50% memory and 75% computation savings with only a 5% accuracy drop. However, the results also showed that the accuracy starts to fall more rapidly when using 3-bit or 2-bit ultralow precision, although this is partly task-dependent. Alternatively, mixed-precision quantization is a method that can be used to optimize a model's weights and activation bit widths separately to the target of the microcontroller's memory and CPU constraints [31]. Furthermore, this method can be used for quantizing each layer separately to different bit widths to maximize accuracy and avoid data loss [32]. Nevertheless, searching for optimal bit widths for all layers is a major computational challenge.

3.2. Binarization. Binarization is another form of quantization, whereby the bit-width compression level is maximized by reducing all operands, weights, and activations to a single bit [33, 34]. In binarized neural networks (BNNs), the arithmetic operations are swapped to bit-wise, and XNOR operations, and only the binarized values (+1 or -1) of the weights and activations are used in all calculations [34]. As a result, 1-bit operations reduce memory need (32) and the number of memory accesses (32) and ultimately lead to increased power efficiency. In [35], the authors introduced embedded binarized neural networks (eBNNs) specially designed for constrained embedded devices. eBNN and BNN have the same network structure, and their model parameters are identical, but they differ in computation order. Computation reordering is needed because original BNNs need a large intermediate pool for storing temporary convolution results in floating-point format, and this pool consumes a large portion of the embedded device's available memory. In eBNN, this is solved with a pool block that can store only one convolution result at a time, and then a max-pooled result is sent through batch normalization and a binary activation function to the result matrix.

3.3. Pruning. An NN's computational complexity can be lowered by pruning its unused features. Pruning techniques can be divided into two main categories: structured and unstructured pruning [36]. Structured pruning means removing entire channels or filters, and unstructured pruning means removing individual weight connections by setting them to zero [37]. In addition, it is possible to combine different pruning approaches. For example, in [38], the authors presented a method whereby unstructured and structured pruning approaches were combined with neural architecture search, which automatically finds accurate, lightweight, and sparse CNN architecture.

The process of zeroing out the NN model's weights is also called magnitude-based pruning, and it leads to a sparse model and can bring a sixfold improvement in model compression [39]. The downside of this method is that it also leads to sparse matrix multiplications that need extra computation power and the use of sparse convolution libraries [40, 41]. Still, by weight pruning the deep neural network (DNN) model's internal redundancy, the model can be downsized, and its performance can be increased without a decrease in the prediction accuracy [40]. The weight pruning method suits the use of microcontrollers in particular because the benefits of model size reduction are more significant than the extra computational load from sparse multiplications.

Structured pruning changes the shapes of layers and weight matrices by removing groups of weight connections [37]. When whole channels or filters are removed, the network's inference speed increases, and the model size decreases. A channel-level pruning produces a lightweight network, but it can lower the model's performance and accuracy when the width of the entire network is reduced. Hence, it is recommended that unstructured pruning methods are used whenever possible. In [40], the authors reported a 3.54-fold mean performance speedup and 88% size reduction in the model when they tested the different weight and node pruning combinations with Arm Cortex-M4 microcontroller with a two-way SIMD (single instruction, multiple data) unit for 16-bit fixed-point mathematics, 128kB SRAM, and 512kB flash storage. In addition, their proposed pruning technique, named Scalpel, a mixture of SIMD-aware weight pruning and node pruning, gained better efficiency and a smaller memory footprint for the model than basic pruning techniques.

3.4. Clustering. The number of individual weight values can be reduced using a process known as clustering, whereby the model's weight values are replaced with a smaller number of centroid weight values that are calculated from the original model's grouped weights [39]. Weight clustering reduces memory usage via model compression, and the compressed

CNN model can be five times smaller than the original. When weight clustering and quantization processes are compared to each other, weight clustering brings higher accuracy and compression ratio, but the two can still be used effectively together [42]. The weight clustering process is typically done with the k-means clustering algorithm [42, 43].

4. Frameworks and Libraries

In real TinyML applications, in addition to the ML model, there is system logic and sometimes a real-time operating system (RTOS) that consumes already limited memory resources [44]; however, in most used cases, the TinyML application does not require RTOS. Nevertheless, RTOSs might sometimes be useful for TinyML applications too, as they are capable of running multithreaded and concurrent software executions [45]. In this kind of multithreaded TinyML application, RTOSs, such as Miosix [46], Zephyr OS [47], Riot OS [48], and Arm Mbed OS can be used [45].

The lack of a unified TinyML framework has led to the use of custom frameworks. Furthermore, custom frameworks, which have limited availability, require complicated manual optimization when used with different hardware. Nevertheless, in the past few years, the TinyML framework development has begun to progress. Among the first frameworks was Arm uTensor, an open-source ML framework for microcontrollers, and then in 2019, uTensor and Google's TensorFlow began to build the TensorFlow Lite for microcontrollers framework together [49]. In recent years, Arm has also released a comprehensive set of network kernels in the software library known as Cortex Microcontroller Software Interface Standard-NN (CMSIS-NN) [50]. Apache, too, has extended its open-source ML framework TVM to cover microcontrollers in μTVM [51]. Another edge ML framework is PyTorch Mobile, which extends the PyTorch ecosystem [52]. In addition to these more versatile frameworks, there is the emlearn library, which is an open-source ML inference engine for microcontrollers starting from 8-bit architecture [53].

4.1. TensorFlow Lite. TensorFlow Lite (TFLite) is an open-source deep learning (DL) framework and set of tools for deploying and running ML models on Android, iOS, embedded Linux devices, and microcontrollers [54]. It supports multiple programming languages, such as Java, Swift, Objective-C, C++, and Python. Nevertheless, when using highly constrained microcontrollers and with only some hundreds or dozens of kilobytes of RAM, TensorFlow Lite for Microcontrollers (TFLM) is an efficient tool to use together with TFLite. TFLM can be used for running ML inference on a device, but it does not yet support on-device training. Its core runtime requires only 16 kB of memory, and it can be used with many Arm Cortex-M architecture microcontrollers. It has also been tested with Espressif ESP32 and different digital signal processors (DSP) [12]. Furthermore, TFLM does not require an operating system, and it can be downloaded as an Arduino library.

The TensorFlow Model Optimization Toolkit can be used to minimize the model's latency, memory utilization, and power consumption. These tools include methods such as post-training quantization (PTQ), quantization aware training (QAT), pruning, and clustering [39]. In addition, TFLite includes TensorFlow Lite converter, which can be used to postquantize an already trained model and convert it to device-optimized TFLite format [55]. Posttraining integer quantization best suits constrained microcontrollers, and the method converts the weight and activation bit-width of 32-bit floating-point numbers to 8-bit fixed-point numbers.

4.2. Cortex Microcontroller Software Interface Standard-NN. The CMSIS-NN library is built for NN development on Arm Cortex-M processors, and inference based on its functions achieves a 4.6-fold speedup in throughput, and a 4.9-fold cut-back in energy consumption [56]. The CMSIS-NN library contains a specific category of NN functions and support functions such as convolution, activation, fully connected layer, pooling, softmax, basic math, activation table, and data-type conversation functions [50]. The functions use either 8-bit or 16-bit integers as parameters, but most of the functions still use 16-bit multiply and accumulate (MAC) instructions for operations such as matrix multiplications [56]. These 16-bit SIMD instructions require an Arm processor with a SIMD unit, but it is possible to use the CMSIS-NN library with older Arm processors such as Arm Cortex-M0 without the SIMD unit [57]. However, Arm Cortex-M0's performance lags behind that of the Arm Cortex-M4, M7, M33, and M35P when using the CMSIS-NN.

4.3. Apache TVM. In recent years, Apache TVM infrastructure has been extended with μTVM, which is software that can manage the host-driven execution of tensor programs on microcontrollers that run without OS [51]. μTVM runtime offers a C-code generator, cross-compiler interface, and μDevice interface as well as interoperability between μTVM runtime and TVM's AutoTVM, an automatic tensor program optimizer. μTVM uses the JTAG (Joint Test Action Group) connection and Open On-Chip Debugger (OpenOCD) control between the target device's processor and the host, i.e., a desktop computer. This setup allows AutoTVM's autotuning process, through which it generates candidate kernels round after round and executes them in the target device; at the end, it uses timing results for autotuning the model parameters. As the results in [51] show, AutoTVM tuning increases performance by lowering the program's execution time from 294 ms to 157 ms, and it is almost the same as the TFLite+CMSIS-NN model.

4.4. PyTorch Mobile. PyTorch Mobile provides simplified end-to-end workflow and execution of ML models on edge devices [52]. It can be used with more powerful mobile operating systems such as iOS, Android, and Linux. PyTorch Mobile includes XNNPACK floating point and QNNPACK 8-bit quantized kernel libraries for mobile-optimized NN inference. PyTorch Mobile cannot be used with the most constrained microcontrollers at this point, but it is possible to use PyTorch models on microcontrollers through Open Neural Network Exchange (ONNX) format conversion with other software, including TensorFlow, STM32Cube.AI, and Cainvas [58–60].

4.5. emlearn. The emlearn library contains a Python-C model converter and inference engine for microcontrollers and other devices that use C-code [53]. It can be used for converting classic ML and NN models such as random forests (RF), decision trees (DT), naive Bayes (NB), multilayer perceptron (MLP), and sequential models built with Keras and scikit-learn frameworks. It supports fixed-point math and does not use dynamic memory allocations. Most of the other discussed frameworks can be mainly used with 32-bit computer architecture, but emlearn can be used with 8-bit AVR processors, as was done by authors in [61]. The emlearn library is similar to MicroMLgen [62], FogML [63], and sklearn-porter [64] libraries.

5. Development Environments

The Edge Impulse, Qeexo AutoML, and Imagimob provide TinyML as a service. The Edge Impulse is an open-source software development kit (SDK) that enables ML on microcontrollers [13], and Qeexo AutoML is an automated ML platform [14]. Another lite toolkit for embedded systems, which we will discuss later in this review, is STMicroelectronics STM32Cube.AI [15]. The Cartesiam NanoEdge AI Studio includes lightweight ML libraries that can be used with all Arm Cortex-M family microcontrollers [65].

5.1. Edge Impulse Studio. The Edge Impulse (EI) SDK can be used for implementing neural networks on embedded devices and includes real sensor data collection and live signal processing, testing, and code deployment to the target device [13]. Furthermore, the actual data can be collected by sensors in IoT devices and mobile phones, and an existing dataset can be uploaded to the EI SDK with an uploader tool in JSON, CBOR, JPG, and WAV formats [66].

The authors in [67] tested the EI SDK in their research to develop a means to triage COVID-19 suspected cases. The authors created a wrist-wearable IoT device based on a 32-bit Espressif ESP8266EX microcontroller. The device was used to measure and process raw photoplethysmogram (PPG) data. These data were extracted to vital components and eventually to 22 NN input features wirelessly transferred to the EI SDK. The patient triage was formed by combining vital PPG components and the EI SDK NN classification toolchain, whereby the classification was made into three classes: slow breathing (bradypnea), normal breathing, and heavy breathing. The selected densely connected pyramid NN architecture gave the model 95.1% accuracy and 138 ms inference time estimation for on-device inference.

5.2. Qeexo AutoML. Qeexo AutoML provides an automated ML platform for Arm Cortex processors and even highly constrained M0 and M0+ processors [14]. Deploying ML on the M0+ can be pretty difficult compared to doing so on an M4 because the M0+ can only calculate 32-bit fixed-point mathematics and has lower memory capacity, lower CPU speed, and no support for saturation arithmetic and DSP [68]. Because of all this, Qeexo AutoML has developed a highly optimized Arm Cortex M0+ fixed-point ML pipeline, including sensor data handling, feature computation, and inference, all with fixed-point data. With the M0+, the pipeline uses tree-based ML algorithms such as gradient boosting machine (GBM), RF, and eXtreme Gradient Boosting (XGBoost). Qeexo AutoML's comprehensive ML algorithm portfolio also includes NB, DT, Isolation Forest (IF), support vector machine (SVM), local outlier factor (LOF), logistic regression (LR), CNN, convolutional recurrent neural network (CRNN), recurrent neural network (RNN), and artificial neural network (ANN) [14]. Qeexo AutoML uses intelligent pruning and posttraining quantization methods resulting in 90% model size compression. Additional 8-bit quantization can shrink the model size by up to 75% compared to models using 32-bit precision [68].

5.3. STM32Cube.AI. STMicroelectronics STM32Cube.AI is an NN and ML toolkit for STM32 developers to run optimized inferences in microcontrollers [15]. STM32Cube.AI tools contain the most common deep learning libraries and decision-making processes with more resource-optimized algorithms such as a DT classifier. The STM32Cube.AI can be expanded with the X-CUBE-AI package, including automatic conversion of pretrained NN and classic ML models. X-CUBE-AI supports all frameworks that use ONNX format, including PyTorch, Microsoft Cognitive Toolkit, and MATLAB, and has support for well-known DL and ML frameworks such as TFlite, Keras, Caffe, Lasagne, ConvnetJS, scikit-learn (IF, SVM, k-means clustering (kMC), etc.), and XGBoost package [58, 69, 70]. In addition, X-CUBE-AI can optimize networks by 8-bit quantization and save weight and activation parameters in external Flash and RAM memories if more extensive networks are used.

5.4. NanoEdge AI Studio. Cartesiam NanoEdge AI Studio comprises software and a collection of AI libraries for embedded developers that can be used as a search engine for choosing an optimal ML algorithm [65]. It includes signal preprocessing, hyperparametrization, anomaly detection, and classification models such as k-nearest neighbor (kNN), SVM, and NN [71]. The NanoEdge AI Studio allows application-specific ML library development, and it enables unsupervised learning, inference, and a prediction that can be run inside a microcontroller [72]. The program automatically tests, optimizes, and calculates the best algorithmic combination as a C library. After the NanoEdge AI Studio has chosen the best library in the project, the library will be able to learn normal behaviors and figure out what an anomaly is [73]. It can perform iterative learning in 30 msecs in an Arm Cortex-M4 80 Mhz and consumes only 4 kB RAM in a typical configuration [74]. It is also worth mentioning that Cartesiam AI has been used in one of the first commercial TinyML products, a sensor called Bob Assistant, which uses automated on-device learning techniques for monitoring machines online [75]. This sensor prepares and sends predictive maintenance reports automatically once the period of learning the machine's normal behavior ends.

5.5. Imagimob. Imagimob has two software products that can be used for building Edge AI applications. The Imagimob AI software suite is an end-to-end development solution for building Edge AI and TinyML applications [76]. It can be used with all types of time-series data, and it focuses on deep learning. Imagimob AI development follows five steps: (1) data capture and labeling, (2) data management in one place, (3) automatic model building with AI training service, (4) model verification with visualization of all models and predictions, and (5) edge optimization and application packaging. Imagimob supports quantization of LSTM (long short-term memory) layers, which is challenging but essential when using time-series data [77]. Edge is the easy-to-operate SaaS solution that can be used for simplifying complex Edge AI and TinyML development [78]. It can convert TensorFlow and Keras h5 file formats into the highly performing C code used in edge devices. This conversion might be a challenging task for even a proficient programmer, but when using Imagimob Edge, it can be done automatically in seconds. The suite is suitable for running DL models on highly constrained embedded devices such as Arm Cortex-M0 microcontrollers with a RAM memory size as small as 10 kB [78, 79].

5.6. TinyML Development Tools Summary. This section summarizes TinyML development tools' features in Table 2, showing the available ML algorithms, supported interoperable frameworks, and the minimum architecture and type of the target processor.

6. TinyML Benchmarking

When discussing the design of a TinyML performance benchmarking test, there are four primary challenges to overcome: (1) varying power consumption across the range of devices; (2) limited and varying memory resources across the range of devices; (3) lack of hardware heterogeneity, which makes it hard to normalize performance results; (4) lack of software heterogeneity because major vendors have their proprietary tools and compilers [5]. In addition to this, the benchmark toolset should cover various ways for model deployment. Today, benchmarking tests are designed to benchmark either ML inference or microcontroller performance rather than the intersection of these technologies. One of the unsuitable benchmarking methods is the MLPerf Inference Benchmark [80], which is targeted at more powerful computers. Recently, the authors of [81] have introduced the MLPerf Tiny Benchmark Suite to meet the requirements of TinyML. This open-source suite [82] can be used to measure the accuracy, latency, and energy consumption of TinyML inference. The MLPerf Tiny v0.5 provides visual wake words, keyword spotting, anomaly detection, and image classification tasks for benchmarking, including reference implementations, which are provided using TFLite and TFLM [81]. The suite can be used for evaluating embedded devices that have clock speed in the range of 10 MHz–250 MHz and which typically consume less than 50 mW per inference [82].

The authors in [61] tested emlearn, sklearn-porter, and MicroMLgen classic ML libraries with an extremely constrained Arduino Uno microcontroller that had only an 8-bit processor, a clock speed of 16 MHz, 32 kB of flash memory, and 2 kB of SRAM. Their work selected DT, RF, SVM, and MLP algorithms for the test, and the benchmark showed that DT and RF gave the best accuracy, lowest memory footprint, and fastest classification speed. The MLP algorithm benchmark test showed good 0.97 accuracy with one hidden layer with four neurons, but its weights and biases did not continue to fit Arduino Uno's SRAM when the network complexity grew. MicroMLgen's SVM was the weakest performing algorithm in the benchmark in terms of accuracy and memory footprint.

7. Emerging Techniques of TinyML

Among the latest emerging TinyML techniques is federated learning (FL), which was introduced and defined in [83]. FL is a large-scale machine learning technique, whereby ML models are trained in remote devices while keeping training data localized [84]. Therefore, FL enables data privacy and security when the attack surface is limited only to the IoT devices themselves [83]. FL architectures can be divided into centralized and decentralized ones [85]. In the centralized approach, there is a server between end devices, and in the decentralized approach, end devices can exchange data between themselves. For example, when centralized edge devices collaboratively train a prediction model, they first update new parameters locally to the shared prediction model, then send updates to the server and finally receive the aggregated model back from the server [86]. In the typical decentralized approach, each device can perform local updates to ML model parameter gradients after the device has received gradient updates directly from all other nodes [87].

One key challenge when combining FL and TinyML techniques is model on-device training, usually not supported in TinyML frameworks. Still, on-device training and evaluation can be implemented with programming languages such as Java, Swift, and C/C++ [86]. Furthermore, FL resource optimization can be done using a technique known as transfer learning (TL), which uses older models to generate a new one [88]. This procedure reduces the computational resources required to train a new model. In [89], the authors presented a method named federated transfer learning on tiny devices (TinyFedTL), whereby they implemented their own fully connected layer inference and backpropagation update between an Arduino Nano 33 BLE Sense microcontrollers and a local server. As a result, they managed to train an ML model without sending raw data to the server; only the weights and bias data had to be sent between the client nodes and the server. Nevertheless, as in any other ML model training procedure, also in the FL approach, the training efficiency and model accuracy depend on the data set quality and computing power [90]. The TL approach is also an effective method to use by itself. Like in [91], the Tiny Transfer Learning (TinyTL) reduced

Table 2: TinyML development tools' features.

Development tool	Algorithms and support	Target processor
Edge Impulse Studio	Proprietary NNs	32-bit (Cortex-M0+–M7)
Qeexo AutoML	GBM, RF, XGBoost, NB, DT, IF, LR, LOF, SVM, CNN, CRNN, RNN, ANN	32-bit (Cortex-M0–M4)
STM32Cube.AI (+X-CUBE-AI)	Scikit-learn (IF, SVM, kMC, etc.), XGBoost, Keras, TFLite, Caffe, Lasagne, ConvnetJS	32-bit (Cortex-M0⟶)
NanoEdge AI Studio	NN, kNN, SVM, proprietary ML	32-bit (Cortex-M0⟶)
Imagimob	Proprietary NNs	32-bit (Cortex-M0⟶)
emlearn	RF, DT, NB, MLP, Keras, scikit-learn	8-bit (ATmega328P⟶)

Table 3: Summary of emerging techniques of TinyML.

Technique	Main features
Federated learning	Edge devices collaboratively train an ML model Improved data privacy and security E.g., TinyFedTL [89], centralized, and decentralized approaches
Transfer learning	Uses older ML models to generate a new one E.g., TinyTL [91]
On-device learning	Uses streaming data for training ML models in a microcontroller E.g., TinyOL [22] and NanoEdge AI Studio[7]
LPWAN	Long-range, low power, and low data rate E.g., LoRaWAN, Sigfox, NB-IoT, and LTE-M

memory footprint up to 6.5-fold. TinyTL uses pretrained models to save the microcontroller's memory resources by not storing activations, learning only biases, and freezing the weights.

Another recent article proposed a method known as TinyML with Online-Learning (TinyOL), which can use streaming data for posttraining and upgrading of the existing on-device NN model [22]. In this method, an extra TinyOL training layer is used interleaved with the prediction phase. After new data first flow through the existing TinyML model to the inference phase and the result label is found, the evaluation metrics and weights are updated according to the new data. When TinyOL uses an incremental learning process, it decreases the microcontroller's memory and processor usage compared to batch learning because new data can be handled one by one, and in the end, the data can be erased when the update is finished. Besides modern NN models, traditional ML algorithms such as NB, SVM, LR, and DT are even better suited for resource-constrained on-device training because their resource demands are typically low [92]. In addition, as in TinyML overall, lowering model complexity with dimensionality reduction and pruning and lowering computational load with quantization helps achieve better on-device training performance.

One of the most attractive emerging technology combinations is the integration of low-power wide-area networks (LPWANs) with TinyML. Energy efficiency and large coverage are the foremost defining characteristics of LPWANs, although they have a low data rate [93]. Therefore, LPWAN radio technologies such as LoRaWAN, Sigfox, NB-IoT, and LTE-M are ideal technology partners for TinyML; this is because in TinyML, the inference is made inside a constrained microcontroller, and in most cases, only a compressed inference result is needed to send to the server. Today, when national LPWAN networks and their coverage are becoming more widespread worldwide, it is even more tempting to avoid separate gateway devices and build stand-alone end-node applications. This makes sense because it is also easier to set up one device than a complex combination of separate end-nodes and gateways. Electronics manufacturers have also discovered this, and they have started to integrate 32-bit Arm Cortex processors and sub-GHz radios in system-on-chip (SoC) and system-in-package (SiP) units [94, 95]. In [85], the author built a wearable TinyML device whereby he integrated the MLP model and peripherals such as a LoRaWAN transceiver, GPS module, and inertial sensor. The results showed that the peripherals' libraries and MLP model's memory footprint were below 2 kB SRAM, which is small enough even for the tiniest microcontrollers. Table 3 summarizes the advantages of emerging techniques of TinyML.

8. Conclusion

TinyML has recently been continuously studied by different organizations, which have in turn created various frameworks, tools, and methods for applying ML on microcontrollers. In these studies, overcoming the microcontrollers' resource constraints has been the main research topic; as presented in many articles, this is typically done through computational means by lowering the memory footprint of an ML model, which also has a positive effect on microprocessors' CPU usage and power consumption. However, the downside in this is the tradeoff between model size and accuracy, as it has an accuracy-lowering impact, although this is at a reasonable level in most cases. Anyhow, TinyML is still in its early stages, and commercial products, for example, are mainly still to be realized. Therefore, the future of TinyML evolution depends on how companies and the academic community focus their resources for testing and benchmarking various TinyML applications and algorithms. A comprehensive benchmark tool that can be used with a range of microcontrollers is a vital first step for creating a continuum for research.

9. Future Application Areas

Overall, when considering the technical evolution of small IoT devices from a broader perspective, there have not yet been any megatrend products that everybody should own. Those tiny IoT devices that are available are used mainly for controlling purposes and for perhaps sending data over the internet. Nevertheless, in the future, TinyML is likely to change the evolution and demand of tiny IoT devices, and we will see many new must-have products in this category. The main reason for this is that new intelligent products are at the center of a digital, data-orientated, energy-efficient, and resource-optimized lifestyle. An excellent example of this product category is wearable technology, which combines health, personal safety, and communication technology. Hence, the future use of TinyML is not limited just to areas where microelectronics are now present, but it will also extend to new fields and inexpensive products. One example of this product category comprises condition monitoring solutions that are currently used only with critical and expensive machinery. Therefore, low-cost TinyML sensors will be likely to extend condition monitoring applications to less critical and mobile machinery that does not even need electricity since the sensor can use a battery. In addition, these kinds of machines are also good targets for FL applications because they are typically mass-produced, and so they could join together to produce an ML model that could be generalized to different situations. Another thing worth considering is how techniques such as TinyML, FL, on-device learning, and LPWAN could influence research in different fields of natural science so that the behaviors of geographically distributed study objects can be classified and observed on-device, and then the inference results or even upgraded parameters of the ML model are sent to the server. In addition, this improves data privacy and security because no sensitive raw data are sent to cloud servers. Inference engine on edge also reduces inference time and network usage, which can be critical features for some applications. From a negative point of view, TinyML may be used for ethically controversial solutions such as military, surveillance, and hacking devices. Thus, it is essential to remember the ethical aspect when building TinyML applications. Finally, TinyML is likely to cement its position among other ML techniques, and its maturity will quickly multiply over time.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work has been done under the eÄlytelli and coADDVA funded by the European Regional Development Fund and the Regional Council of Central Finland.

References

[1] M. Shirer and C. MacGillivray, *The Growth in Connected IoT Devices is Expected to Generate 79.4 ZB of Data in 2025, According to a New IDC Forecast*, IDC, 2019.

[2] A. Yousefpour, C. Fung, T. Nguyen et al., "All one needs to know about fog computing and related edge computing paradigms: a complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[3] E. Raj, "What is edge computing and EdgeAI?," 2021, https://www.tietoevry.com/en/blog/2020/03/what-is-edge-computing-and-edgeai/.

[4] tinyML Foundation, "TinyML," 2021, https://www.tinyml.org.

[5] C. R. Banbury, V. J. Reddi, M. Lam et al., "Benchmarking TinyML systems: challenges and direction," 2021, http://arxiv.org/abs/2003.04821.

[6] H. Han and J. Siebert, "Tinyml: a systematic review and synthesis of existing research," in *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pp. 269–274, Jeju Island, Korea, 2022.

[7] STMicroelectronics, "The Onlife era of MEMS: integrating AI in sensors for decision-making in the edge," 2022, https://www.st.com/content/st_com/en/campaigns/ispu-ai-in-sensors.html.

[8] Bosch, "BHI260AP ultra-low power, high performance, self-learning AI smart sensor with integrated accelerometer and gyroscope," 2022, https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bstbhi260ap-ds000.pdf.

[9] P. Andrade, I. Silva, M. Silva, T. Flores, J. Cassiano, and D. G. Costa, "A TinyML soft-sensor approach for low-cost detection and monitoring of vehicular emissions," *Sensors*, vol. 22, no. 10, p. 3838, 2022.

[10] T. Flores, M. Silva, P. Andrade et al., "A TinyML soft-sensor for the internet of intelligent vehicles," in *2022 IEEE InternationalWorkshop on Metrology for Automotive (MetroAutomotive)*, pp. 18–23, Modena, Italy, 2022.

[11] S. A. Manzano, V. Sundaram, A. Xu et al., "Toward smart composites: small-scale, untethered prediction and control for soft sensor/actuator systems," 2022, http://arxiv.org/abs/2205.10940.

[12] R. David, J. Duke, A. Jain et al., "Tensorflow lite micro: embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[13] Edge Impulse, "TinyML for all developers with Edge Impulse," 2020, https://www.hackster.io/news/tinyml-for-alldevelopers-with-edge-impulse-2cfbbcc14b90.

[14] Qeexo, "Enabling the new era of machine learning at the edge," 2021, https://qeexo.com/.

[15] STMicroelectronics, "STM32Cube.AI: convert neural networks into optimized code for STM32," 2020, https://blog.st.com/stm32cubeaineural-networks/.

[16] Y.-L. Lee, P.-K. Tsung, and W. Max, "Techology trend of edge AI," in *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1-2, Hsinchu, Taiwan, 2018.

[17] Partha Pratim Ray, "A review on TinyML: stateof-the-art and prospects," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 4, pp. 1595–1623, 2022.

[18] F. Ventures and H. Collins, "Why TinyML is a giant opportunity," 2021, https://venturebeat.com/2020/01/11/why-tinymlis-a-giant-opportunity/.

[19] ABI Research, "Global shipments of TinyML devices to reach 2.5 billion by 2030," 2020, http://www.abiresearch.com/press/global-shipmentstinyml-devices-reach-25-billion-2030/.

[20] TechAheadCorp, "How TinyML can transform IoT applications across industries," 2021, https://www.techaheadcorp.com/blog/tinymltransform-iot-applications/.

[21] J. Pena Queralta, T. N. Gia, H. Tenhunen, and T. Westerlund, "Edge-AI in LoRabased health monitoring: fall detection system with fog computing and LSTM recurrent neural networks," in *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, pp. 601–604, Budapest, Hungary, 2019.

[22] H. Ren, D. Anicic, and T. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," in *2021 International Joint Conference on Neural Networks (IJCNN)*, Shenzhen, China, 2021.

[23] K. Dokic, M. Martinovic, and D. Mandusic, "Inference speed and quantisation of neural networks with tensorflow lite for microcontrollers framework," in *2020 5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDACECNSM)*, pp. 1–6, Corfu, Greece, 2020.

[24] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: keyword spotting on microcontrollers," 2017, http://arxiv.org/abs/1711.07128.

[25] Wikipedia, "ARM Cortex-M," 2020, https://en.wikipedia.org/wiki/ARM_Cortex-M.

[26] A. Developer, "Arm software development toolkit reference guide," 2022, https://developer.arm.com/documentation/dui0041/c/Floating-point-Support/About-floating-point-support?lang=en.

[27] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," 2015, http://arxiv.org/abs/1412.7024.

[28] C. Zhang, "How to run deep learning model on microcontroller with CMSIS-NN (part 3)," 2018, https://www.dlology.com/blog/howto-run-deep-learning-model-on-microcontrollerwith-cmsis-nn-part-3/.

[29] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of The 33rd International Conference on Machine Learning*, pp. 2849–2858, New York, New York, USA, 2016.

[30] S. Zhuo, H. Chen, R. K. Ramakrishnan et al., "An empirical study of low precision quantization for TinyML," 2022, http://arxiv.org/abs/2203.05492.

[31] M. Rusci, M. Fariselli, A. Capotondi, and L. Benini, "Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers," in *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning. ITEM IoT Streams 2020*, vol. 1325 of Communications in Computer and Information Science, pp. 296–308, Springer, Cham, 2020.

[32] H.-A. Rashid, P. R. Ovi, C. Busart, A. Gangopadhyay, and T. Mohsenin, "Tinym2net: a flexible system algorithm codesigned multimodal learning framework for tiny devices," 2022, http://arxiv.org/abs/2202.04303.

[33] L. Mocerino and A. Calimera, "Fast and accurate inference on microcontrollers with boosted cooperative convolutional neural networks (BC-Net)," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 1, pp. 77–88, 2021.

[34] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: training deep neural networks with weights and activations constrained to +1 or -1," 2016, http://arxiv.org/abs/1602.02830.

[35] B. McDanel, S. Teerapittayanon, and H. T. Kung, "Embedded binarized neural networks," 2017, http://arxiv.org/abs/1709.02260.

[36] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–18, 2015.

[37] M. Kurtz, "Part 1: what is pruning in machine learning?," 2020, https://neuralmagic.com/blog/pruning-overview/.

[38] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough, "Sparse: sparse architecture search for CNNs on resource-constrained microcontrollers," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[39] Google, "TensorFlow model optimization," 2020, https://www.tensorflow.org/model_optimization/guide.

[40] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: customizing DNN pruning to the underlying hardware parallelism," *SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.

[41] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2017, http://arxiv.org/abs/1608.08710.

[42] S. Ye, T. Zhang, K. Zhang et al., "A unified framework of DNN weight pruning and weight clustering/quantization using ADMM," 2018, http://arxiv.org/abs/1811.01907.

[43] L. Meng and N. Suda, "Optimizing Power and Performance for Machine Learning at the Edge: Model Deployment Overview," *ARM AI - AI Virtual Tech Talks Series*, pp. 1–35, 2020.

[44] C. Banbury, C. Zhou, I. Fedorov et al., "Micronets: neural network architectures for deploying tinyml applications on commodity microcontrollers," in *Proceedings of machine learning and systems*, pp. 517–532, San Jose, CA, USA, 2021.

[45] O. S. Mbed, "Features and benefits of Mbed OS," 2021, https://os.mbed.com/mbed-os/.

[46] F. Alongi, N. Ghielmetti, D. Pau, F. Terraneo, and W. Fornaciari, "Tiny neural networks for environmental predictions: an integrated approach with Miosix," in *In 2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 350–355, Bologna, Italy, 2020.

[47] Zephyr, "Introduction," 2022, https://docs.zephyrproject.org/latest/introduction/index.html.

[48] O. S. Riot, "Welcome to the friendly operating system for the Internet of Things," 2021, https://www.riot-os.org.

[49] N. Tan, P. Warden, and Z. Shelby, "uTensor and TensorFlow announcement," 2019, https://os.mbed.com/blog/entry/uTensor-and-Tensor-Flow-Announcement/.

[50] Arm, "CMSIS NN software library," 2021, https://arm-software.github.io/CMSIS_5/NN/html/index.html.

[51] L. Weber and A. Reusch, "TinyML -How TVM is taming tiny," 2021, https://tvm.apache.org/2020/06/04/tinyml-how-tvm-is-taming-tiny.

[52] PyTorch, "PyTorch Mobile," 2021, https://pytorch.org/mobile/home/.

[53] J. Nordby, *emlearn: Machine Learning Inference Engine for Microcontrollers and Embedded Devices*, GitHub, 2019, https://github.com/emlearn/emlearn.

[54] Google, "TensorFlow Lite: deploy machine learning models on mobile and IoT devices," https://www.tensorflow.org/lite.

[55] Google, "Model optimization," 2020, https://www.tensorflow.org/lite/performance/model_optimization.

[56] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: efficient neural network kernels for ARM Cortex-M CPUs," 2018, http://arxiv.org/abs/1801.06601.

[57] Arm, "Image recognition on ARM Cortex-M with CMSIS-NN," 2021, https://developer.arm.com/solutions/machine-learning-on-arm/developermaterial/how-to-guides/image-recognition-onarm-cortex-m-with-cmsis-nn/single-page.

[58] STMicroelectronics, "Artificial intelligence (AI) software expansion for STM32Cube," 2020, https://www.stmicroelectronics.com.cn/resource/en/data_brief/x-cube-ai.pdf.

[59] R. Sharma and P. Jain, "Bringing PyTorch models to ARM Cortex-M processors," 2021, https://developer.arm.com/solutions/machinelearning-on-arm/community/ai-virtual-techtalks.

[60] A. Singh, "Converting a model from Pytorch to Tensorflow: guide to ONNX," 2021, https: //http://analyticsindiamag.com/converting-a-modelfrom-pytorch-to-tensorflow-guide-to-onnx/.

[61] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-enabled frugal smart objects: challenges and opportunities," *IEEE Circuits and Systems Magazine*, vol. 20, no. 3, pp. 4–18, 2020.

[62] "MicroMLgen," 2021, https://github.com/eloquentarduino/micromlgen.

[63] T. Szydlo, J. Sendorek, and R. Brzoza-Woch, "Enabling machine learning on resource constrained devices by source code generation of the learned models," in *International Conference on Computational Science*, vol. 10861 of Lecture Notes in Computer Science, , pp. 682–694, Springer, 2018.

[64] D. Morawiec, "sklearn-porter," 2021, https://github.com/nok/sklearn-porter.

[65] Cartesiam, "Cartesiam: leader in edge AI market, with proven industrial reference," 2020, https://cartesiam.ai.

[66] Edge Impulse, "Documentation," 2021, https://docs.edgeimpulse.com/docs.

[67] B. Fyntanidou, M. Zouka, A. Apostolopoulou et al., "IoT-based smart triage of COVID-19 suspicious cases in the emergency department," in *2020 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6, Taipei, Taiwan, 2020.

[68] R. Bhatt and T. Shyuan, "Building effective IoT applications with TinyML and automated machine learning," 2021, https://www.embedded.com/building-effective-iot-applications-withtinyml-and-automated-machine-learning/.

[69] STMicroelectronics, "AI expansion pack for STM32CubeMX," 2020, https://www.st.com/en/embedded-software/x-cube-ai.html.

[70] STMicroelectronics, "X-CUBE-AI documentation," 2022, https://wiki.st.com/stm32mcu/wiki/AI:X-CUBE-AI_documentation.

[71] STMicroelectronics, "NanoEdge AI Studio," 2022, https://wiki.st.com/stm32mcu/wiki/AI:NanoEdge_AI_Studio.

[72] M. Vetrano, "Cartesiam AI development environment brings artificial intelligence, learning and inference to everyday objects," 2020, https://www.prweb.com/releases/cartesiam_ai_development_environment_brings_artificial_intelligence_learning_and_inference_to_everyday_objects/prweb16933237.htm.

[73] Cartesiam, "Frequently asked questions:input data and formatting," 2020, https://cartesiam-neaidocs. http://readthedocs-hosted.com/faq.html.

[74] Design and Reuse, "Cartesiam transforms edge AI development for industrial IoT," 2020, http://www.design-reuse.com/news/49170/cartesiamnanoedge-ai-studio-ide-arm-cortex-m-mcu.html.

[75] nkeWATTECO, "BoB Assistant," 2021, https://bobassistant.-com/en/offer/#talents.

[76] Imagimob AB, "Imagimob AI," 2021, https://developer.imagimob.com/#/./imagimob-ai.

[77] J. Malm, "Quantization of LSTM layers- a technical white paper," 2022, https://www.imagimob.com/blog/quantizationof-lstm-layers-a-technical-white-paper.

[78] Imagimob AB, "Introducing Imagimob Edge: Making Tensorflow AI models edge device ready at the click of a button," 2020, https://www.imagimob.com/news/introducingimagimob-edge-making-tensorflow-ai-modelsedge-device-ready-at-the-click-of-a-button.

[79] EDGE Computing World, "Edge startup of the year CXO interviews: Anders Hardebring, CEO and co-founder Imagimob AB," 2020, https://www.edgecomputingworld.com/2020/09/01/startup-of-the-year-finalist-anders-hardebring/.

[80] V. J. Reddi, C. Cheng, D. Kanter et al., "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459, Valencia, Spain, 2020.

[81] C. Banbury, V. J. Reddi, P. Torelli et al., "MLPerf tiny benchmark," 2021, http://arxiv.org/abs/2106.07597.

[82] N. Jeffries, C. Kiraly, C. Banbury et al., "MLPerf tiny deep learning benchmarks for embedded devices," 2021, https://github.com/mlcommons/tiny.

[83] H. Brendan McMahan, E. Moore, D. Ramage, S. Hampson, and B. Aguera y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pp. 1273–1282, Fort Lauderdale, FL, USA, 2017.

[84] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: challenges, methods, and future directions," *IEEE Signal Processing*, vol. 37, no. 3, pp. 50–60, 2020.

[85] R. Sanchez-Iborra, "Lpwan and embedded machine learning as enablers for the next generation of wearable devices," *Sensors*, vol. 21, no. 15, p. 5218, 2021.

[86] A. Mathur, D. J. Beutel, P. P. B. de Gusmão et al., "On-device federated learning with flower," 2021, http://arxiv.org/abs/2104.03042.

[87] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief, "Communication-efficient edge AI: algorithms and systems," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2167–2191, 2020.

[88] M. M. Grau, R. P. Centelles, and F. Freitag, "On-device training of machine learning models on microcontrollers with a look at federated learning," in *Proceedings of the Conference on Information Technology for Social Good*, pp. 198–203, New York, NY, USA, 2021.

[89] K. Kopparapu and E. Lin, "TinyFedTL: federated transfer learning on tiny devices," 2021, http://arxiv.org/abs/2110.01107.

[90] Q. Miao, H. Lin, X. Wang, and M. M. Hassan, "Federated deep reinforcement learning based secure data sharing for Internet of Things," *Computer Networks*, vol. 197, article 108327, 2021.

[91] H. Cai, C. Gan, L. Zhu, and S. Han, “Tiny transfer learning: towards memory-efficient on-device learning,” 2020, http://arxiv.org/abs/2007.11622.

[92] S. Dhar, J. Guo, J. Liu, S. Tripathi, and U. Kurup, *On-DeviceA Survey of On-Device Machine Learning: An Algorithms and Learning Theory Perspective*, 2020.

[93] Z. Ahmad, S. Jahari, F. Zaman, S. A. R. Al-Haddad, and A. Sali, *LPWAN State of the Art: Trends and Future Directions*, ResearchGate, 2021.

[94] STMicroelectronics, “Ultra-low power multi-modulation wireless STM32WLE5x microcontrollers,” 2020, https://www.st.com/en/microcontrollersmicroprocessors/stm32wlex.html.

[95] Microchip, “ATSAMR34J18,” 2020, https://www.microchip.com/wwwproducts/en/ATSAMR34J18.