AIfES: A Next-Generation Edge AI Framework

Lars Wulfert, Johannes Kühnel, Lukas Krupp, Justus Viga, Christian Wiede, Pierre Gembaczka, and Anton Grabmaier

***Abstract*—Edge Artificial Intelligence (AI) relies on the integration of Machine Learning (ML) into even the smallest embedded devices, thus enabling local intelligence in real-world applications, e.g. for image or speech processing. Traditional Edge AI frameworks lack important aspects required to keep up with recent and upcoming ML innovations. These aspects include low flexibility concerning the target hardware and limited support for custom hardware accelerator integration. Artificial Intelligence for Embedded Systems Framework (AIfES) has the goal to overcome these challenges faced by traditional edge AI frameworks. In this paper, we give a detailed overview of the architecture of AIfES and the applied design principles. Finally, we compare AIfES with TensorFlow Lite for Microcontrollers (TFLM) on an ARM Cortex-M4-based System-on-Chip (SoC) using fully connected neural networks (FCNNs) and convolutional neural networks (CNNs). AIfES outperforms TFLM in both execution time and memory consumption for the FCNNs. Additionally, using AIfES reduces memory consumption by up to 54% when using CNNs. Furthermore, we show the performance of AIfES during the training of FCNN as well as CNN and demonstrate the feasibility of training a CNN on a resource-constrained device with a memory usage of slightly more than 100 kB of RAM.**

***Index Terms*—Edge AI Framework, embedded systems, machine learning framework, on-device training, resource-constrained devices, TinyML.**

I. Introduction

OVER recent years, Machine Learning (ML) has become one of the main drivers of innovation in engineering and scientific applications [1], [2]. With currently estimated more than 250 billion embedded devices in use [3], this trend also extends to embedded systems bringing ML-enabled computing capabilities closer to the data sources. Often referred to as edge Artificial Intelligence (AI) or TinyML, these methods have recently found their way into microcontroller units (MCUs), which make up the Internet of Things (IoT). According to [4], using TinyML offers numerous improvements compared to cloud AI solutions in terms of data protection, low processing latency, energy saving, and minimal connectivity dependency. An extensively researched application for TinyML based on artificial neural networks (ANNs) pertains to condition monitoring utilizing intelligent sensor systems (e.g. [5], [6], [7], [8]). The primary objective is to detect anomalous machine behavior directly within the sensors themselves. This approach enables the transmission of only anomalous data, thereby reducing energy consumption as transmitting data is more energy-intensive compared to local processing [9], [10]. Furthermore, the sensor system can be directly connected to the machine's control system, allowing for immediate intervention in the event of a defect. Consequently, there is no need for data to be sent to a cloud server for defect detection, highlighting the data protection aspect of TinyML and emphasizing the reduced processing latency. Additionally, this system can be deployed in remote areas with limited or no connectivity.

To develop efficient and effective ML methods, numerous frameworks are available that utilize high-performance computing hardware, like graphic processing units (GPUs). PyTorch [11] and TensorFlow [12] are two of the most popular and widely used frameworks in this context.

However, since the hardware resources of embedded systems are often very restricted, these frameworks cannot be used. Therefore, specialized tools and frameworks have been developed that allow migrating of ML models, trained on high-performance hardware using large datasets, to resource-constrained devices. Such traditional edge AI frameworks, like TensorFlow Lite for Microcontrollers (TFLM) [13] or Apache TVM [14], focus mainly on the inference and optimization of a wide variety of ANNs, enabling the deployment of deep neural networks (DNNs) on embedded platforms like MCUs. However, converting the ML models between frameworks and even programming languages is unavoidable since the training was done on a PC. Furthermore, the frameworks are often bound to specific hardware platforms with limited possibilities of integrating specialized hardware acceleration. As a result, developing or training an ANN directly on a resource-constrained system is impossible.

Therefore, several developments in the domain of on-device training of ML models have been carried out in the last two years [15], [16], [17], [18], [19], [20]. On-device training of ML models such as support vector machine (SVM) [21], k-nearest neighbor (K-NN) [19] and decision trees (DT) [16] were made feasible first. However, SVM and K-NN have the drawback that the training data must be retained in order to make robust predictions, resulting in high memory requirements [22]. In addition, while SVM, K-NN, as well as DT can learn linear

Manuscript received 20 January 2023; revised 23 October 2023; accepted 13 January 2024. Date of publication 18 January 2024; date of current version 7 May 2024. Recommended for acceptance by M. Sugiyama. *(Lars Wulfert, Johannes Kühnel and Lukas Krupp contributed equally to this work.) (Corresponding authors: Lars Wulfert; Johannes Kühnel; Lukas Krupp.)*

Lars Wulfert, Johannes Kühnel, Lukas Krupp, Christian Wiede, Pierre Gembaczka, and Anton Grabmaier are with the Fraunhofer Institute for Microelectronic Circuits and Systems, 47057 Duisburg, Germany (e-mail: lars.wulfert@ims.fraunhofer.de; johannes.kuehnel@ims.fraunhofer.de; lukas.krupp@ims.fraunhofer.de; christian.wiede@ims.fraunhofer.de; pierre.gembaczka@ims.fraunhofer.de; anton.grabmaier@ims.fraunhofer.de).

Justus Viga is with RWTH Aachen University, 52076 Aachen, Germany (e-mail: justus.viga@rwth-aachen.de).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPAMI.2024.3355495, provided by the authors.

Digital Object Identifier 10.1109/TPAMI.2024.3355495

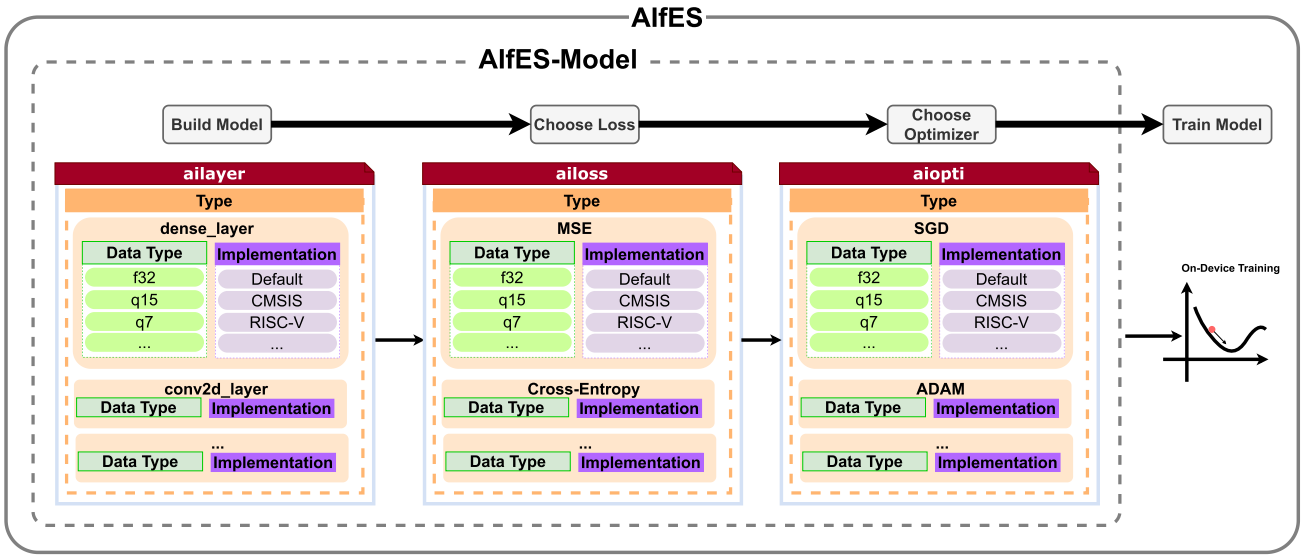


Fig. 1. Structure of AIfES to build a model and perform on-device training includes four steps: (1) building the model, (2) selecting the loss, (3) choosing the optimizer, and (4) training the model. The AIfES model is built from a customizable structure composed of the modules *ailayer*, *ailoss*, and *aiopti*. Each module consists of the hierarchy layers *Type* (functions of the module), *Implementation* (use of hardware accelerators), and *Data type* (data type with which the ML models are executed).

relationships very effectively, they reach their limits when non-linear correlations are learned. In contrast, ANN are able to acquire and train nonlinear relationships in data and are therefore suited for complex problems [23]. As the number of embedded systems continues to rise, there is also increasing interest in having more complex tasks, such as condition monitoring, predictive maintenance or object detection in images, performed on MCUs [20]. Due to restricted resources of MCUs, on-device training of fully connected neural network (FCNN) and convolutional neural networks (CNNs) was considered unfeasible until recently [24]. However, preliminary work has shown that on-device training of FCNN [15] and CNN [20] on MCUs is feasible. Although the frameworks and methods are compatible with different MCUs such as RISC-V, Cortex-M, or ESP32, many lack open access and modular software structure so that customized functions can be integrated by the user, e.g., activation function. Even though training FCNNs and CNNs on MCUs is now feasible, complex arithmetic operations (e.g., matrix multiplication) are still performed that are challenging for MCUs. With customized hardware accelerators, complex calculations are performed faster which saves resources and energy [25], [26], [27]. However, none of the existing developments enable a modular structure to insert custom hardware accelerators in their framework.

We introduce Artificial Intelligence for Embedded Systems (AIfES),[1] a hardware-independent edge AI framework that bridges the gap between resource-constrained embedded systems and sophisticated machine learning models. As depicted in Fig. 1, the modular structure of AIfES is designed to follow the well-known structure of ML frameworks, such as Keras or PyTorch. This structure includes four steps: (1) building the model, (2) selecting the loss, (3) choosing the optimizer and (4) training the model. This approach enables users to easily transfer their experiences to AIfES. Furthermore, the *ailayer* module of AIfES offers a selection of different function types, such as *dense layer*. Users can assign a data type, such as 8-bit, to each function. This feature can help to save memory or enable faster training or inference on devices without floating-point unit (FPU). Moreover, AIfES is the first framework that provides the ability to use hardware accelerators in a modular fashion within an ML framework. The software's modular design allows for the addition of new user-specific hardware accelerators, function types, data types, or modules. This allows customization of the framework according to the user's needs and preferences. Leveraging C and optimized modules, AIfES enables on-device training and inference of ML models without requiring an operating system. To perform the inference, AIfES requires only the structure and weights of a pre-trained ANN model. Optimized modules and custom hardware accelerators can be easily integrated to enhance the inference or training performance of the model. ANNs can be loaded, fine-tuned, or the network structure can be changed at runtime. Even training from scratch is possible without pre-training, avoiding the need to send training data to a more powerful and energy-consuming device. Energy is saved without sharing the raw training data, and privacy increases. Furthermore, on-device training enables a new generation of self-learning systems and sensors that adapt to new data and can even be combined on-demand using Federated Learning (FL) [28] to increase performance further. The main contributions of this paper are:

- AIfES an open source edge AI framework that provides inference and on-device training for resource-constrained devices that is hardware agnostic and software modular

[1] [Online]. Available: https://github.com/Fraunhofer-IMS/AIfES_for_Arduino

- The framework supports the modular addition of user-specific hardware accelerators to enhance the performance of inference and on-device training. It also includes software optimization modules for activation functions
- Modular framework that allows network architectures to be adapted or changed at run time
- We conducted an extensive evaluation to demonstrate the framework's effectiveness and variety of settings. Starting with FCNNs for inference, comparing our framework with TFLM using hardware accelerators, quantization, and on-device training. Furthermore, we evaluated CNNs on-device training on well-known datasets

The remainder of this paper is organized as follows. In Section II, we provide background on on-device learning frameworks and an overview of related work. Subsequently, we present our proposed framework, AIfES, in Section III. Furthermore, detailed insight is provided into the design principles, such as modular architecture, memory usage, and hardware and software optimizations for reduced runtime. Section IV the inference is evaluated for different datasets and network structures for FCNNs and CNNs. Also, an analysis of on-device training of AIfES is presented. Finally, Section V summarizes the paper and provides an outlook on future work.

II. Background and Related Work

Due to the enormous potential of and interest in edge AI and TinyML, the number of frameworks, libraries and tools is constantly growing [4], [29]. The most commonly used conversion approach relies on deploying pre-trained ML models on embedded platforms, like MCUs. Consequently, well-known ML libraries such as TensorFlow [12], Scikit-Learn [30] or PyTorch [11] are used to create the model and train it. Subsequently, the pre-trained ML model can be used on the resource-constrained system. Frameworks such as TFLM have been developed to apply the models to MCUs. TFLM optimizes TensorFlow models to run efficiently on mobile and embedded devices. Utility functions are provided to reduce the size and complexity of an ML model. Several ANN architectures are supported, which can be used for inference on different platforms after conversion. These include smartphones, embedded Linux systems and 32-bit MCUs. [13]. Edge Impulse [31] is a service that uses a completely different approach to pre-train ML models and deploys them on the edge device. First, the data must be uploaded to the cloud, where the training will be performed. Afterward, the ML model can be converted to various libraries of the required hardware, such as C++, Arduino, or Cube.AI library. Even conversion to WebAssembly and binary files are supported. Subsequently, the library can be deployed on smartphones, CPU/GPU, or a variety of supported MCUs, e.g., Nordic Semi nRF52840 DK. TFLM is used to run the Edge Impulse ML models on the resource-constrained devices [31].

There are also manufacturer-specific solutions that operate according to a similar principle. A first example is the STM32Cube.AI [32] toolkit for STM32 ARM Cortex-M-based MCUs and their X-Cube-AI [33] extension for optimizations. The toolkit can convert pre-trained ANNs from TensorFlow, Keras [34] or models in ONNX [35] format into C code. With NanoEdge AI Studio [36] it is also feasible to incrementally train the ML models on STM32 MCU. Another tool is Microsoft's Embedded Learning Library (ELL) [37], which enables the development and deployment of pre-trained ML models on resource-constrained platforms, such as Arm Cortex-A and Cortex-M-based architectures. ELL is an optimized cross-compiler that runs on a regular desktop computer and outputs MCU-compatible C++ code [29].

Several techniques have been developed to address TinyML's low-resource challenges, including pruning [38], [39], [40], [41], [42], [43], [44], Quantization [38], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54] and neural architecture search (NAS) [52], [55], [56], [56], [57], [58], [59], [60], [61], [62]. These methods reduce model parameters while maintaining model accuracy, allowing the models to be applied to MCUs. Although quantization is supported in the framework, other optimizations such as pruning or NAS are not yet available. Instead, the framework emphasizes a modular software architecture and the modular addition of custom hardware accelerators. This modular approach allows adding pruning or NAS methods individually, as described in Chapter III-A.

In addition to the frameworks capable of performing inference only, there were developments on on-device training of ML models using MCUs. Table I depicts a detailed list of publications. In the review, 19 publications were identified in which on-device training was conducted. The majority support the ARM Cortex family, and [15], [19], [21], [24], [63], [64], [65] support even multiple MCU families such as ESP32 or AVR MCUs. The publications used the programming languages C or C++ to implement the ML algorithms since these languages are suitable for hardware-near implementations and offer fast execution times with low memory requirements [66].

There have been many successful attempts to train ML algorithms on MCUs, besides from ANNs a variety of algorithms such as SVM, DT, RF or K-NN have been applied and shown to allow training of these algorithms on MCUs. These publications include Edge2Train [21] and Train++ [24], which use SVM for on-device training, whereas [16] only supports training of K-NN or DT. Other papers such as [19], [22], [65] support further algorithms such as SVM in addition to FCNN or CNN. For instance, [22] compares the memory requirements and inference time of different algorithms for TinyML, including FCNN, SVM, RF, LR, GNB, and DT. Lee et al. [19] proposed an intermitted learning framework for energy-harvested computing platforms supporting unsupervised and semi-supervised learning algorithms. Although they publish their framework and support a software modular architecture, they have neglected to support hardware accelerators, which can save energy [25], which is particularly relevant in energy-saving systems operated by energy harvester [19]. Almost 80 % of the publications in the field of microcontrollers and on-device training address FCNN or CNN, since these methods can train complex nonlinear correlations, allowing more complex applications of ML methods on MCUs [23]. There are seven recent publications [17], [18], [67],

TABLE I
COMPARISON BETWEEN AIFES AND VARIOUS PAPERS WITH ON-DEVICE TRAINING USING MCU. $\surd$ = TRUE, $\times$ = FALSE, - = NOT IDENTIFIED

Paper	Compatible MCU	Language	ML-Model	Open Source	Modularity	
					Software	Hardware
[3]	nRF52840	C/C++	FCNN, CNN	×	-	-
[17]	Cortex-M	C/C++	FCNN	×	-	-
[18]	Cortex-M4	C++	FCNN	×	-	-
[67]	Cortex-M	C	FCNN	×	-	-
[68]	Cortex-M7	C	FCNN	×	-	-
[22]	nRF52840	C	DT, Random Forest (RF), Logistic Regression (LR), Gaussian Naive Bayes (GNB), FCNN SVM	×	-	-
[69]	Cortex-M3	C	CNN	×	-	-
[65]	STM32	C	FCNN, CNN, K-NN, SVM	×	-	-
[63]	STM32	C	CNN	×	-	-
[64]	Adafruit Feather, STM32, ESP32, Adafruit METRO	C	-	✓	×	×
[24]	Xtensa, ESP32, Cortex-M	C++	SVM	✓	×	×
[70]	Arduino Uno	C/C++	FCNN	✓	×	×
[21]	ESP32, Cortex-M	C++	SVM	✓	×	×
[71]	Cortex-M4	C++	FCNN	✓	×	×
[72]	Arduino Portenta H7	C/C++	FCNN	✓	×	×
[16]	Cortex-M	C	K-NN, DT	✓	✓	×
[19]	AVR, PIC, MSP430	C	K-NN, K-Means, FCNN	✓	✓	×
[20]	Cortex-M7	C	CNN	✓	✓	×
[15]	RISC-V, STM32	C	FCNN, CNN	✓	✓	×
AIfES	All GCC compatible MCU (e.g., Cortex-M, Arduino, STM32, Atmel AVR, etc.)	C	FCNN, CNN	✓	✓	✓

[68], [70], [71], [72] that have successfully performed on-device training using only FCNNs. In [17], on-device training is investigated and compared using an Arduino Nano 33 BLE Sense and an Arduino Protenta H7, resulting in the Portenta H7 training a FCNN 4.2 times faster. Incremental learning is performed for supervised and unsupervised learning using an autoencoder in [18]. To use a self-adaptive control algorithm for a DC motor controller, [67] trains a FCNN from scratch. An autoencoder without anomalies was initially trained in [68] to monitor the condition of rotating machines on MCU. The model was then used to detect anomalies in the machines afterward. With the increasing interest in FL where different devices collaborate to train ML models, several methods have been proposed for MCUs [3], [71], [72], [73]. Ren et al. [3] proposed a federated meta learning approach for resource constrained-devices. However, they did not specify whether they used an existing library or developed the on-device training from scratch. In [72] and [73] algorithms from [70] are included. In contrast, the training in [71] was developed from scratch since no available frameworks supported it yet [71]. Similarly, due to a lack of support in available frameworks, [69] proposed a method for training CNNs on MCUs. Additionally, Lin et al. [20] proposed a method for training CNNs models with less than 256 KB random access memory (RAM) including two key innovations for on-device training through Quantization-Aware Scaling (QAS) and Tiny Training Engine (TTE). First, QAS stabilizes training by automatically scaling the gradient of tensors with different bit-precisions. Second, TTE optimizes the runtime by performing auto-differentiation and sparse update at compile-time. Although Lin et al. [20] demonstrated the feasibility of on-device training with less than 256 KB RAM, they pre-trained the models and performed post-training quantization in their experiments before running fine-tuning on MCUs. Therefore, it is uncertain whether training under these resource constraints would be possible without pre-training and post-quantization (i.e., training from scratch). An optimization was also proposed

with Parallel-Ultra-Low-Power (PULP) [15], using a RISC-V-based parallel software approach. Also, they propose a strategy to automatically select the fastest kernel depending on the tensor shapes in each DNN layer. In addition to FCNN, PULP is also capable of training CNN in parallel on different RISC-V cores. Opt-SGD and Opt-OVO are presented as optimization methods for binary and multiclass ML classifier training in the paper [64]. In [65] and [63], the X-Cube-AI [33] extension of STM is used for on-device training, while in [65] the Forward-Forward [74] training method is used for the first time on MCUs, where no backpropagation is required. De Vita et al. [63] use the extension to train an echo state network [75], which is a form of recurrent neural network (RNN), on an STM32 board.

Although a total of 19 publications have been published on the topic of on-device training on MCUs, the source code has only been published for slightly half of the papers, so validation or further development of the work is not possible, which slows down the acceleration of the research area and optimization of the algorithms. Furthermore, slightly more than half are available with open access, whereas only [15], [20], [19] uses a modular software structure. A modular software structure is shown by the implementations being systematically divided into logical sub-blocks.

We noticed that there are some promising solutions with [20], [65], [19] and [15] to deal with little resources while training not only FCNN but also CNNs and RNN. However, none of the current work includes a modular hardware structure allowing users to include their hardware accelerator into the framework. A modular hardware structure implies that arbitrary hardware accelerators can be added to the framework as long as they are callable by the used programming language. In addition, training is developed from scratch in several publications such as [17], [67], [68], [69], [71]. Thus, we conclude that there is a need for a modular TinyML framework for on-device training on resource-constrained devices. To address this gap, we present AIfES.

To overcome this gap, the open source framework AIfES, presented in this paper, has been developed. It targets all types of MCUs ranging from small 8-bit MCUs up to powerful e.g., ARM Cortex-M-based MCUs and supports both inference and on-device training of FCNN and CNN. However, AIfES can as well be used on a PC for evaluation or visualization purposes.

III. Design Principles

AIfES is specifically designed to run on embedded, low-resource devices like MCUs. Therefore, the requirements differ compared to usual machine learning frameworks. A major goal of AIfES is to make the usage of the library as simple and intuitive as possible while being efficient enough to run even on the smallest MCUs and flexible enough to support most of the use cases in ML. Therefore, we propose a modular cuboid structure of AIfES depicted in Fig. 2, which is designed to provide a flexible and customizable structure in which users can individually select the available functions from the modules *ailayer*, *ailoss* and *aiopti*. Each module in itself has different hierarchy layers *Type* (functions of the module), *Implementation* (use of hardware accelerators) and *Data type* (data type with which the ML models are executed), which allows different settings and configurations. For instance, in the *ailayer* module, users can select different layers, such as Dense or Conv2D layer, to be included in their ML model. The cuboid structure allows the user to extend the existing modules and hierarchy levels or even add new ones as required.

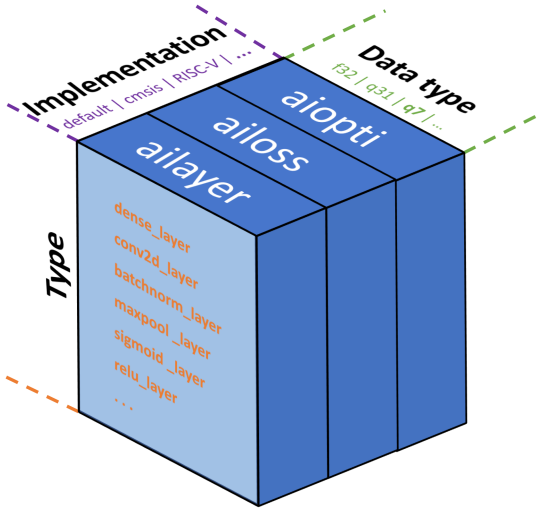


Fig. 2. Modular and customizable cuboid structure of AIfES composed of the modules *ailayer*, *ailoss* and *aiopti*. Each module consists of the hierarchy layers *Type* (functions of the module), *Implementation* (use of hardware accelerators) and *Data type* (data type with which the ML models are executed).

A. Modular Architecture

The AIfES framework has a modular architecture. An ANN model can be built out of processing blocks called layers that are connected to form the whole model, which is also employed by the commonly used modern deep learning frameworks like Keras [34] and PyTorch [11]. This allows experienced users of Keras or PyTorch to use AIfES more easily. For training, loss functions are assigned to the model, and it can be trained with different optimizers to perform the gradient steps of the backpropagation algorithm. Unlike other frameworks, AIfES also takes the data type and the underlying system particularities in the foreground, which are essential factors on resource-constrained devices. Moreover, it provides all components required for training an ANN right on the device, like backward implementations of all layers, several loss functions and optimizers, and weight initialization functions. An overview of the supported components is given in the appendix. These components follow the same modular concept and are flexible and adaptable to any system and use case.

Fig. 3 shows the hierarchical structure of modules in AIfES. Every *category* (e.g. layer, loss, optimizer) contains specific modules (e.g. Dense layer, ReLU layer) that define the functionality of the module. Each module *type* can work on data of several *data types* (e.g. float32, int8). The final *implementation* can then be system-specific (e.g. Arm Cortex M, AVR ATMega) to get optimal performance on any hardware.

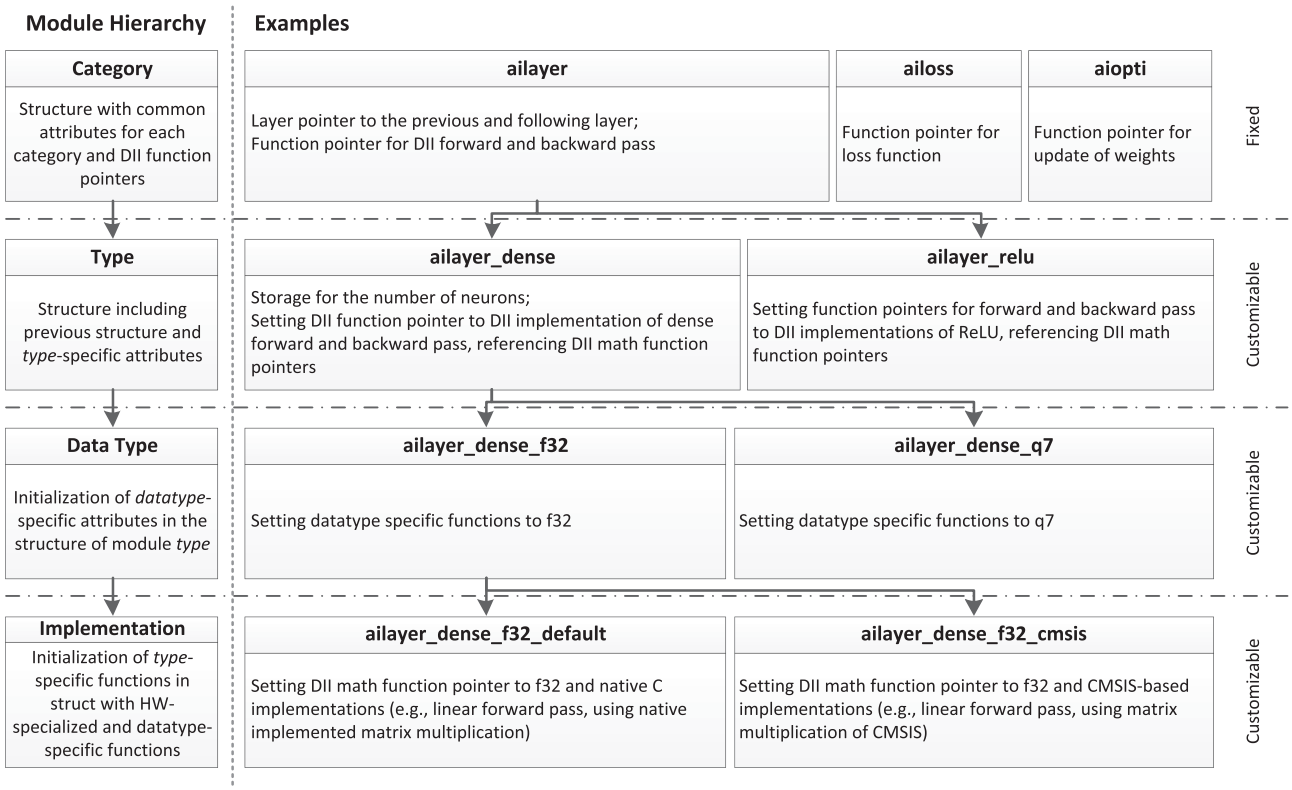


Fig. 3. Structural concept of the modules in AIfES. Arrows are indicating sub-types, similar to inheritance in object-oriented programming. On the left side the hierarchy is given, while the right side gives some examples for an implementation of the hierarchy. The examples describe what type of parameters are set at each level of the hierarchy, to allow software and hardware flexibility.

Thereby, the higher modules pass on general properties to the modules below them. This is done by using structures, which are part of the structures of the lower-level modules. In the case of a layer, the *category ailayer* contains common attributes like a pointer to the previous and following layer. Furthermore, some function pointer are provided. These function pointer are called during a forward or backward pass of the ANN and represent an abstract (*data type*- and *implementation*-independent (DII)) call location during the passes. In contrast, the *ailoss* and *aiopti categories* contain abstract loss- and optimization-specific attributes, respectively (e.g., function pointer for loss calculation for *ailoss* or function pointer for parameter update for *aiopti*).

The module *type* describes common attributes which are DII but specific to their operation. This allows for different functions, e.g., dense layer in contrast to activation function layer for *ailayer* or different loss functions for *ailoss*. For the example of a dense layer, the module *type* provides arguments like the number of neurons, tensor pointers for weights and bias, and initializes the function pointers from the *category* for the forward and backward pass with DII function implementations for dense layers. In the case of an activation layer, the function pointer for the forward and backward pass are initialized with DII versions of the activation layer. Those DII functions use the underlying mathematical functions to implement the desired functionality (e.g., tensor add and multiply operations or matrix multiplication). Hereby, the mathematical functions are also DII, as they are referenced as function pointers from the final *implementation* module.

Consequently, the *data type*-specific representation initializes the data type of the layer. Combined with the final *implementation* module, the DII function pointers are initialized with *data type*- and *implementation*-dependent versions. All the needed mathematical functions are provided by a separate mathematics module, where the needed functions are referenced for the initialization of the DII function pointers. The mathematics module contains implementations for each *data type* and *implementation*, e.g., the matrix multiplication of the forward pass in f32 and q7 *data type*. AIfES currently offers two different types of *implementations*. The default implementation is purely software-based and is tested on various systems to provide the best performance in most cases. In contrast, the Common Microcontroller Software Interface Standard (CMSIS) implementation uses the CMSIS digital signal processor (DSP) functions for an efficient implementation on Arm Cortex-M MCUs by optimizing the implementation of the mathematical functions.

In order to use a different hardware accelerator (in the shape of existing or custom-designed hardware units), the hardware-optimized mathematical functions (like tensor multiplication) need to be added to the mathematics module. Additionally, a final *implementation* must be added to the desired layer. In the final *implementation* the specialized mathematical functions need to

be referenced. No further adjustments are necessary, as the DII implementations of the modules *type* and *category* automatically use the mathematical functions given in the *implementation*. With this design concept, a hardware developer does not need to know about neural networks to develop hardware accelerators. Instead, a machine learning expert can use accelerated building blocks provided by the hardware expert. Moreover, this allows for a hardware/software co-design workflow where the developer starts with the default implementations and gradually replaces them with custom accelerated functions. An example for customized hardware accelerators can be found in [25]. In this example, custom single-instruction-multiple-data (SIMD) instructions were integrated into an RISC-V MCU to improve the calculation of dense layers. Here, only the *implementation* of the dense layer needed to be updated. For this, the function pointer for the default implementation needs to be replaced with the SIMD specific implementation. No further changes are necessary, highlighting the modular structure of AIfES. With this concept, a hardware developer only needs to develop a mathematical function for matrix multiplication, which is then referenced by the function pointer inside the *implementation*. Furthermore, the activation functions were optimized with custom hardware accelerators, where also only the function pointers needed to be updated to automatically use the optimized hardware accelerators.

Moreover, easy porting of the framework to other hardware architectures is possible. As the framework is entirely written in C and is compatible with the GNU Compiler Collection (GCC), the default *implementation* is executable on any hardware that is supported by the GCC and customized hardware accelerators can be included as described above.

With this modular concept, adding new components and adapting to new use cases is straightforward without diving deep into the framework code, as seen by integrating new hardware accelerators. Additional *types* can be added, e.g., to support new activation functions, where the additional mathematical functions need to be added and referenced in the new final *implementation*. The clear design choice of structuring AIfES into the different modules *data type* and *implementation* leads to a more efficient system, as unnecessary functions can be excluded during implementation, compilation and, therefore, during deployment.

B. Memory

A main constraining factor on embedded devices is limited memory. On the one hand, the RAM for storing variables and mutable data is often only a few kilobytes (or even a few bytes) of size and must therefore be used very sparingly. On the other hand, the read-only memory (ROM) for program code storage and constant variables is also essential. This contrast with non-edge devices, where the code size is often not considered. With its modular design, AIfES makes it easy for C-compilers to remove unused code and thus shrink the code size. The memory for the parameters (e.g. weights) and intermediate results of an ANN can be individually assigned depending on the application. Constant parameters, like non-trainable parameters, can be stored in ROM (e.g. Flash memory or EEPROM) or external storage components, while mutable parameters (e.g. gradients, errors, momentum, quantization parameters) need to consume space in RAM. For instance, gradients, errors, intermediate results, and quantization parameters will be placed in RAM.

Unlike a dynamic memory allocation like Keras [34], or a custom caching memory allocator such as PyTorch [11], AIfES uses a different approach. As a primary design concept, all the memory is assigned before running the network. For this, AIfES provides scheduler functions to calculate the required memory size beforehand based on the network structure of the FCNN and distribute a block of memory to the model.

First, the memory size for the inference of each layer is calculated. Thereby, the number of mutable parameters is multiplied by the selected data type. The memory size for the intermediate results of the quantization parameters is also determined if quantization is used. For this, the size of the used data type is added to the memory block. Subsequently, the address sections of the memory block are added to the individual layers depending on the number of variable parameters by the scheduler. Afterward, the memory size for the training is identified if the FCNN should be trained by AIfES. Hereby, the memory size for the intermediate results of the different layers for the forward- and backward-pass is computed. The memory size for the forward and backward pass is determined by the data type and the most significant number of weights and biases in the FCNN. Furthermore, the memory size of the gradients and optimization memory (e.g. first or second momentum) is ascertained. The size of the gradients is determined by the size of the tensor and the used data type. The memory size for the optimizer depends on the chosen one, as Adam in contrast to SGD needs additional storage for the moments. In addition, if applied, the memory size of the quantization parameters is calculated by the utilized data type. After calculating the size of the memory block, the scheduler allocates the address ranges of each layer based on the size of the mutable parameters, optimization size, and memory size for the intermediate results.

Thus, AIfES has no internal dynamic memory allocation (apart from local variables on the call stack). This ensures that the system can not run out of memory during inference or training of an ANN, which is particularly important in safety-critical applications like autonomous driving. Furthermore, no memory fragmentation can occur because the memory scheduler knows when and how much memory is needed during runtime and can optimize the assignment.

C. Hardware and Software Optimizations for Reduced Runtime

Keeping the runtime of ML inference and training low is a key objective of AIfES. A major portion of the execution time of state-of-the-art neural networks is devoted to matrix operations, e.g. in fully-connected or convolutional layers. However, comparably small neural networks are frequently used on embedded systems due to prevailing resource constraints. With decreasing network size, the activation layers become increasingly relevant concerning their contribution to the execution time. Therefore,

AIfES includes runtime-optimized activation functions and layers in addition to the matrix multiplication-based layers. As several activation functions require the calculation of the exponential function, that can be costly in its default implementation, AIfES includes an optimized variant [76]. Furthermore, AIfES employs piecewise linear approximation (PLA) of activation functions that introduce minor calculation errors but speed up their execution. Hence, AIfES allows adapting the degree of approximation depending on the precision and runtime constraints of the application. AIfES does not use look-up table-based activation functions to prevent a further increase of the library's memory requirements.

For the backward pass of the model, no automatic differentiation is executed. Consequently, a separate implementation of the backward pass is provided for every layer. Thus, no additional bookkeeping of the executed functions on a tensor is necessary, resulting in reduced storage and computing requirements.

AIfES provides two complementary backpropagation workflows to achieve lower memory consumption during training. Both commence with a forward pass that retains the results. The traditional approach progresses by iterating over all layers in reverse, computing and storing the gradients at each step. All parameters are only updated at the very end of this process. The lightweight stochastic gradient descent (L-SGD) algorithm [77] operates differently, retaining only the partial derivatives necessary for the subsequent layer and directly updating the parameters using the calculated gradients. Thus, it only keeps two layers in memory at any given moment.

We have expanded this algorithm to include other optimizers such as ADAM. Additionally, we have made it possible to use the lightweight backpropagation workflow with batch learning, enhancing its practical utility. In this context, we accumulate the gradients over a complete batch in each iteration and update before advancing to the next layer. The lightweight procedure becomes increasingly efficient as the depth of the model increases. For larger batch sizes, the algorithm is quicker due to memory access, though this comes at the expense of higher peak memory usage. AIfES provides users with the flexibility to select the workflow that best suits their network architecture and performance requirements.

Another factor to be aware of when developing a library for embedded purposes is the huge range of underlying hardware configurations. On systems with only 8-bit memory bandwidth and no FPU, the optimal implementation of an algorithm is different than on 32-bit systems with SIMD instructions or DSP accelerators. The modular concept of AIfES allows the development of individual components that fit perfectly to the given hardware platform and instruction set (cf. Section III-A and Fig. 3). For example, the CMSIS for ARM-based MCUs allows acceleration of the calculations by utilizing, among other optimizations, SIMD instructions. The CMSIS can be used in AIfES by the additional CMSIS-implementation.

AIfES allows quantizing your model. Quantization enables an adaptation for different hardware architectures, e.g. to the named 8-bit MCU with no FPU. A quantized ANN to Q7 can improve the calculations. Two quantizations (Q7 and Q31) are offered as a symmetric 8-bit/32-bit integer quantization facilitates integer-only calculations on real values by following the proposed techniques of [50].

IV. Evaluation

In order to evaluate the performance of AIfES a benchmark including multiple ANN architectures (FCNNs and CNNs) and datasets was developed. The performance of AIfES in terms of execution time and memory consumption is compared to TFLM for the inference of the models and AIfES is evaluated in a training scenario.

A. Benchmark Setup

The experiments were conducted on the nRF52840 DK (ARM Cortex-M4-based) by Nordic Semiconductor [78]. The nRF52840 System-on-Chip (SoC) runs at a clock rate of 64 *MHz*.

For the software development and programming of the SoC, the PlatformIO IDE [79] was used. For the compilation, the GCC included in the GNU ARM Embedded Toolchain [80] was used with maximum optimization (-O3). The execution time of inference and training was measured with a logic analyzer (Digital Discovery by Digilent) and the results were evaluated statistically. In the following, only the mean execution time is reported, as the deviations were insignificant. The given values for the memory consumption in terms of RAM and flash memory were taken from the compilation report of PlatformIO. For the inference setting, the parameters of the ANNs were declared with the *const* classifier to place them in the flash memory of the SoC during compile time. The same procedure for the inference and training experiments was used to place the input data in the flash memory for AIfES and TFLM, respectively. However, the input data size was subtracted from the reported flash memory consumption, as only the storage requirements of the two frameworks should be compared and the size of input data varies with the different ANNs. The benchmarks were conducted with the two data types F32 and Q7 for the FCNNs and only F32 for the CNNs. For the Q7-based versions, the pre-trained model from Keras was quantized.

To be able to control the behavior of TFLM, the official repository from GitHub [81] was downloaded and the provided converter tool was used to create the library for ARM Cortex architectures with and without optimized CMSIS kernels. The library is then included in the PlatformIO IDE. For TFLM, pre-trained ANNs from TensorFlow needs to be converted to a TensorFlow Lite model. The converted models are then exported and included in the benchmarking environment. The size of the *kTensorArenaSize* was estimated empirically for each ANN, as it contains all necessary parameters for the ANN and therefore changes size with each tested ANN. A conversion of the pre-trained models from Keras [34] to AIfES is executed. Only the weights and bias are transferred to AIfES to convert a pre-trained model. For both AIfES and TFLM, version 5.8.0 of CMSIS is used [82].

TABLE II
SUMMARY OF INFERENCE EXPERIMENTS WITH FCNNS

Experiment	Dataset	# of inputs & outputs	Hidden layers
1	Iris[85]	4 features, 3 classes	1 hidden layer with 10 neurons (1 x 10)
			1st hidden layer with 10 neurons, 2nd with 50 neurons (10 + 50)
			10 hidden layers with 10 neurons each (10 x 10)
2	Breast Cancer [86]	30 features, 2 classes	1 x 10
			10 + 50
			10 x 10
3	MNIST [84]	64 features, 10 classes	1 x 10
			10 + 50
			10 x 10
4	MNIST [84]	784 features, 10 classes	32 + 32 + 16 (FCDNN 1)
			128 + 64 + 32 + 16 (FCDNN 2)

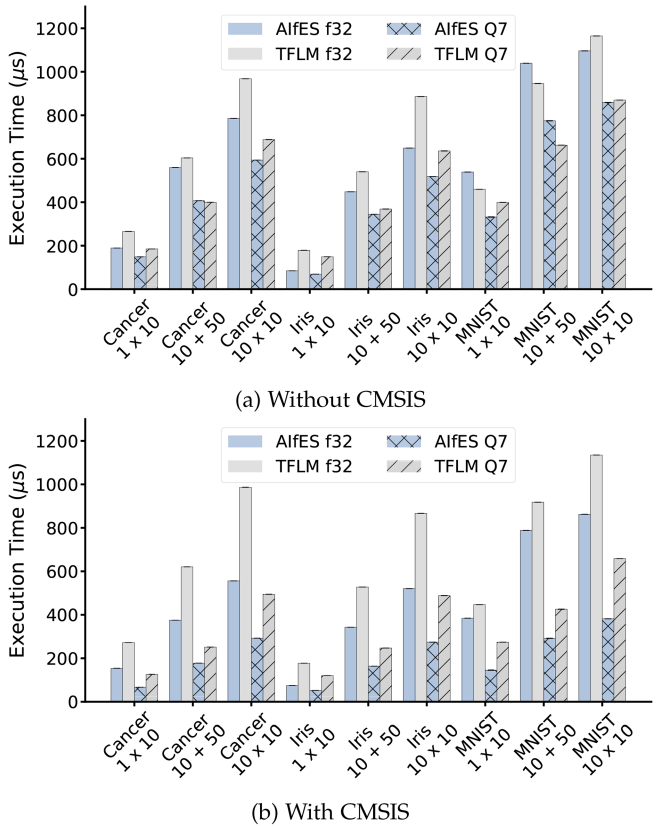


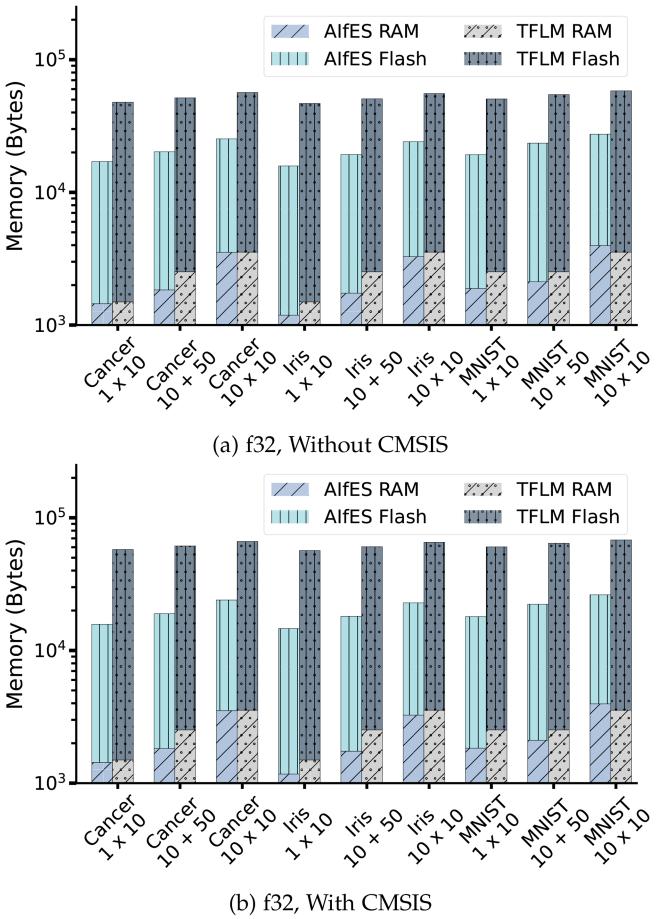
Fig. 4. Comparison of the execution times between AIfES and TFLM for different FCNNs (experiments 1 to 3) is shown, with comparisons between the F32 and Q7 versions illustrated in both subfigures. (a) Represents the inference time with the standard implementation, and (b) with the CMSIS implementation.

B. Inference Benchmark

1) FCNNs: For the evaluation of the FCNNs, the model architectures from an existing TinyML benchmark [83] were adopted. Three representative datasets with several numbers of input features (4 - 64 features) were selected. Additionally, two larger fully connected deep neural network (FCDNN) architectures were evaluated based on the MNIST dataset [84] using the complete and flattened images (784 features). The experiments with the corresponding evaluated datasets and models are summarized in Table II. Experiments 1 to 3 were conducted first. The results are shown in Figs. 4, 5 and 6.

Fig. 4(a) shows that the execution time of the AIfES models exceed that of the TFLM models in most of the cases. Without CMSIS, a speed-up by factors of up to 2.1 for F32 respectively 2.2 for Q7 was measured. The execution times of the slower F32-AIfES models (MNIST 1x10 and MNIST 10 + 50) lie within 17% of that of the TFLM models. At the same time the Q7-based version of MNIST 1x10 is slightly faster (speed-up by factor 1.2), whereas the Cancer 10 + 50 model is slightly slower (by 2 %). An explanation for the lower performance of the MNIST F32 ANNs might be an optimized matrix multiplication implementation of TFLM, taking effect for fully connected layers with a higher number of parameters. This fits with the results for the Q7 Cancer 10 + 50, as for the Cancer 10 + 50 the execution time from AIfES is slightly longer than TFLM. At the same time the optimizations by Q7 allows AIfES to be slightly faster than TFLM in the Q7 MNIST 1 x 10 setting. Fig. 4(b) shows that the AIfES models with CMSIS are faster than the TFLM models in all cases by factors of up to 2.4 and 2.3 for F32 respectively Q7. The slow performance of the F32-based versions of TFLM can be attributed to the fact that TFLM uses the default implementation for F32 without using any function from the CMSIS. This leads to almost the same results as for F32 without CMSIS. For the Q7 setting, TFLM uses CMSIS-based implementations, showing a performance increase, also compared to the Q7 implementations without CMSIS by factors of up to 1.6 (mean 1.4). At the same time, TFLM can reduce the execution time of Q7 ANNs further with CMSIS in comparison without it by factors of up to 2.7 (mean 2.1). These results demonstrate the effectiveness of the modular and open AIfES architecture, enabling the integration of arbitrary accelerated or optimized implementations of ANN functionalities.

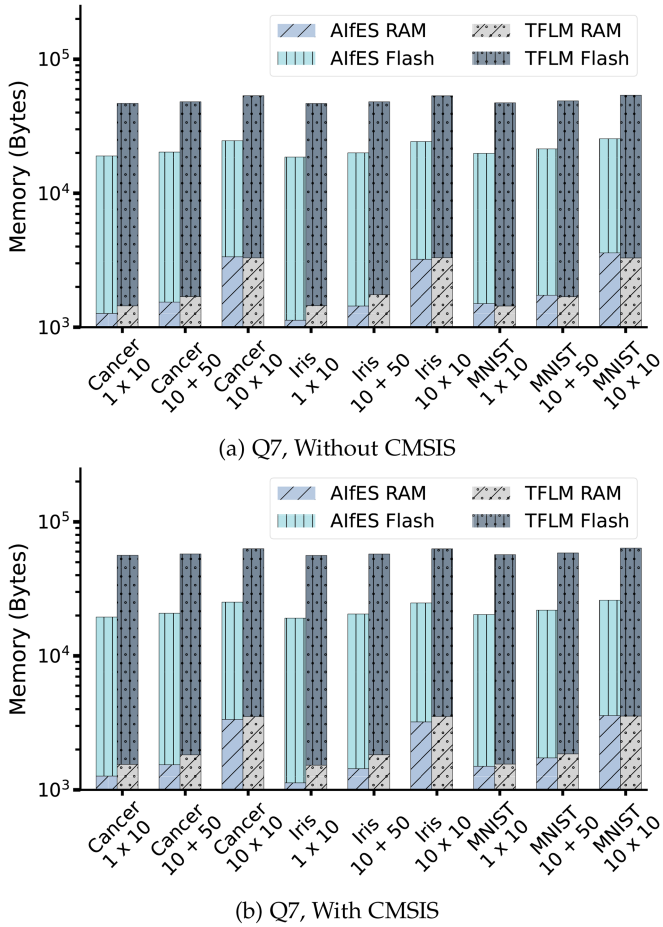
Figs. 5 and 6 show that the AIfES models require overall less memory than the TFLM models by factors of up to 3.9 (starting by factors of 2.1 with a mean of 2.7). The RAM requirements are similar for both frameworks in most cases, while the significant difference is due to the flash memory consumption. We attribute this result to the memory-efficient implementation of AIfES concerning program code and constant variables, typically placed inside the flash memory. Furthermore, it is to note that the flash memory consumption increases for the TFLM models with CMSIS enabled, while it slightly decreases for the F32 AIfES models. The reason for this is that AIfES integrates only a subset



(a) f32, Without CMSIS

(b) f32, With CMSIS

Fig. 5. Memory comparison for the frameworks AIfES and TFLM in the F32 version using different FCNNs (experiments 1 to 3) is shown. Furthermore, the RAM and flash consumption is depicted for the standard and CMSIS implementation. (a) Shows the standard implementation, and (b) illustrates the implementation with CMSIS.



(a) Q7, Without CMSIS

(b) Q7, With CMSIS

Fig. 6. The memory comparison for the frameworks AIfES and TFLM in the Q7 version using different FCNNs (experiments 1 to 3) is shown. Furthermore, the RAM and flash consumption is depicted for the standard and CMSIS implementation. (a) Shows the standard implementation, and (b) illustrates the implementation with CMSIS.

of the CMSIS modules. This can also be seen in the Q7 based implementation, where the flash memory consumption increases for both frameworks but the amount of increase is larger for TFLM. TFLM either uses more or other CMSIS modules with an increased code size or more constant variables. An explanation for the decrease in memory consumption for the F32 AIfES models is the higher efficiency of the CMSIS functions in terms of code size compared to the native AIfES implementations.

Table III shows the results of the FCDNN architectures (experiment 4 in Table II). The execution time of the TFLM models without CMSIS is 17% and 16% lower than that of the AIfES models. This result supports our hypothesis concerning the optimized native matrix multiplication implementation of TFLM for fully-connected layers with higher numbers of parameters. With CMSIS, the execution time of the AIfES models exceed that of the TFLM models by factors of 1.4 and 1.3, respectively. Nevertheless, the RAM requirements of the TFLM models are slightly lower (12% for FCDNN 1 and 5% for FCDNN 2). The flash memory consumption of the AIfES models fall below that of the TFLM models by the same absolute differences as in the previous experiments 1 to 3 (31 kB and 41 kB on average without and with CMSIS respectively). Overall, these results prove the effectiveness of the AIfES architecture again for the integration of external optimized ANN modules and its memory efficiency with respect to flash memory storage.

2) CNNs: Subsequently, 2D-CNN architectures using the MNIST, CIFAR-10 [87] and Visual Wake Words (VWW) [88] dataset were evaluated. The network architecture changes with every dataset since the datasets have a different number of input channels and also a different amount of outputs. For the CIFAR-10 and the VWW, an input of $3 \times 32 \times 32$ was used since the datasets contain RGB images. The images of the VWW were previously resized to fit the input shape of the CNN. The input of $1 \times 28 \times 28$ was used for the MNIST dataset. However, the basic architecture is the same for all of them. Each network has two convolutional layers, the first layer using four kernels and the second layer eight kernels with a size of 3×3 using no padding and a stride of one. ReLU is used in both layers, and maxpooling is performed with a kernel of 2×2 after each convolutional layer. As the CMSIS support for CNNs is not yet included in AIfES, only the native implementations were compared.

TABLE III
EVALUATION OF EXECUTION TIME AND MEMORY CONSUMPTION OF THE FCDNNS (EXPERIMENT 4)

NN Arch.	Framework	Execution Time (ms)		Memory (*kB*)			
		No CMSIS	CMSIS	No CMSIS		CMSIS	
				RAM	Flash	RAM	Flash
FCDNN 1	AIfES	14.47	**8.67**	10.99	**122.28**	10.98	**121.11**
	TFLM	**11.97**	12.40	**9.69**	153.03	**9.69**	162.84
FCDNN 2	AIfES	58.20	**40.71**	11.22	**461.09**	11.21	**459.92**
	TFLM	**49.13**	50.95	**10.71**	491.84	**10.71**	501.66

TABLE IV
EVALUATION OF EXECUTION TIME AND MEMORY CONSUMPTION OF THE 2D-CNN

Dataset	Framework	Execution Time (ms)	Memory (*kB*)	
			RAM	Flash
MNIST	AIfES	51.16	27.01	**34.35**
	TFLM	**43.53**	**16.58**	66.63
CIFAR-10	AIfES	111.03	43.33	**38.53**
	TFLM	**70.85**	**32.96**	70.44
VWW	AIfES	122.42	43.30	**28.48**
	TFLM	**70.40**	**32.96**	61.19

The results in Table IV show that the TFLM CNN exceeds the AIfES model in terms of execution time and RAM requirements by factors of up to 1.74 (VWW) and 1.63 (MNIST) respectively in the worst case evaluations. An explanation for the difference in execution time is the optimized implementation of matrix multiplications in TFLM. AIfES currently uses simple direct convolutions. Sophisticated methodologies such as general matrix multiply (GEMM)- or fast Fourier transform (FFT)-based implementations are not yet included. Similar to the previous experiments, the flash memory consumption of the AIfES CNN is lower than that of the TFLM model with a maximum absolute difference of 32 kB for all datasets. As a result, the flash memory occupancy increases up to 53% in the worst case for the VWW dataset. The significant differences are related to the flat buffer used to store the weights, network structure, and activation functions so that the neural network can be built at run time of TFLM.

TABLE V
EVALUATION OF THE ON-DEVICE TRAINING WITH AIFES OF THE FCNN ON THE NRF52840 DK (BATCH SIZE: 5)

Dataset	Architecture	Training Time/ Batch (ms)	Memory (*kB*)	
			RAM	Flash
Breast Cancer	1 x 10	15.04 (±0.01)	8.34	20.96
	10 + 50	40.34 (±0.02)	21.27	21.22
	10 x 10	55.74 (±0.02)	29.85	23.33
Iris	1 x 10	03.62 (±0.00)	3.54	20.96
	10 + 50	05.41 (±0.01)	17.45	21.22
	10 x 10	07.76 (±0.01)	24.58	23.21
MNIST	1 x 10	33.43 (±0.02)	18.25	21.36
	10 + 50	74.57 (±0.04)	34.80	21.63
	10 x 10	76.67 (±0.04)	39.06	23.74

C. Training Benchmark

Subsequently, we investigated the on-device training for the FCNN and CNN with AIfES. For the evaluation of the FCNN, the same architectures were used for experiments 1 to 3 of the inference benchmark shown in Table II. For the CNNs, the same architectures were chosen as for the inference benchmarks in Section IV-B2. All models were trained using a cross-entropy loss and the Adam optimizer with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ using $\epsilon = 1e^{-7}$ for FCNN and $\epsilon = 1e^{-5}$ for CNNs. Only the training with the default implementation (i.e., without CMSIS) is shown. We measured the execution time per epoch and normalized it by dividing it by the number of batches per epoch, as the batch size is the same (5) for each model.

1) FCNNs: Table V shows that the training of FCNNs on resource-constrained embedded devices is possible with AIfES while keeping the execution time and memory consumption at an acceptable level. This means that the platform is not fully utilized, and enough resources remain. For example, on the experimental platform based on the nRF52840 SoC, 15% of the RAM and 2% of the flash memory are utilized by AIfES in the worst evaluated case (MNIST 10 × 10). Hence, other tasks such as communication, sensor sampling, or signal pre-processing can also run on the embedded system. The overall training

TABLE VI
EVALUATION OF THE ON-DEVICE TRAINING WITH AIFES OF THE 2D-CNN ON THE NRF52840 DK (BATCH SIZE: 5)

Dataset	Training Time/Batch (s)	Memory (*kB*)	
		RAM	Flash
MNIST	1.19	104.14	39.60
CIFAR-10	2.61	149.90	38.80
VWW	2.56	103.64	38.48

execution time of the evaluated models on the experimental platform lies in the range of milliseconds to seconds.

Despite these results, it has to be considered that the model architecture and especially the memory requirements of the training process can be the main limitations of the on-device training with AIfES. We also included a training of a complex deep autoencoder in the appendix B.

2) CNNs: The results of the on-device training benchmark of the CNNs can be seen in Table VI, where we used the same datasets as for the inference evaluations. Also, similar architectures as in the inference analyses were used. The architectures were extended by adding a batch normalization layer after both convolutional layers with $momentum = 0.9$ and $\epsilon = 1e^{-6}$, which accelerates the training according to [89]. Compared to the evaluations of the FCNNs, the training time per batch and the RAM consumption has increased significantly. The CNN trained with the CIFAR-10 dataset takes the most time to train, with 2.61 seconds, and the RAM memory consumption with almost 150 kB. The training time and the memory consumption of the RAM have increased since the number of trainable parameters also increased in the CNNs. For instance, the CNN used in the analysis with the CIFAR-10 dataset uses four times the amount of parameters as the FCNN with the MNIST dataset using 10×10 architecture. Hence the memory requirement is about 3.8 times larger.

The benchmark shows that the training of CNNs is feasible with AIfES, but the training time, as well as the memory consumption, goes up compared to the FCNN training. Nevertheless, the training time remains within an acceptable time frame. However, training deep CNN would be challenging. In addition, the amount of data required to train a deep CNN cannot be stored directly on an MCU.

D. Approximation Benchmark

Since AIfES uses approximations, an important aspect is to examine the relative error of these approximations. For this purpose, the Deep Autoencoder included in the TinyML Perf Benchmark [90] was used. The ReLU activation functions were replaced with sigmoid activation functions to enable an integrated approximation of the activation function in AIfES. First, the autoencoder was trained in Tensorflow. Then, the model was exported to Tensorflow Lite and AIfES. These two models were then executed on the PC, and the mean squared error from the input value was calculated. Based on the 2459 test datasets, the reference values were determined using Tensorflow. Afterwards, the same test data was used to measure the mean squared error of the autoencoder using Tensorflow Lite and AIfES. Subsequently, the mean relative error of Tensorflow Lite and AIfES with respect to the reference values from Tensorflow was determined. Tensorflow Lite has a mean relative error of 1200.95588 ‰, while AIfES has an error of 38.4283 ‰. Thus, the mean relative deviation of AIfES is negligible. The applied approximations in an application example lead to no significant difference in the pAUC (-0.027) value compared to Tensorflow and to minimal difference in the mean value of the AUC of -0.032.

V. CONCLUSION & FUTURE DIRECTIONS

In this paper, we presented the next-generation edge AI framework AIfES. It is specifically designed to leverage the full potential of ML on resource-constrained embedded devices. Compared to other traditional edge AI frameworks, AIfES not only supports the inference on embedded systems but also the on-device training. This allows the use of FL and online learning (OL) techniques in real-world applications. Furthermore, due to its modular architecture, AIfES enables the easy integration of arbitrary optimized and hardware-accelerated ANN functionalities. We performed benchmarks comparing AIfES to TFLM in multiple inference scenarios on an ARM Cortex-M4-based SoC. Especially for FCNN architectures, we showed that AIfES is capable of outperforming TFLM in terms of execution time and memory consumption. Furthermore, we demonstrated the feasibility of the training of ANNs and CNNs on embedded devices with AIfES. The current main limitation of AIfES is the implementation of the native matrix multiplication, leading to a lower performance of ANNs compared to TFLM. In the future, we will enhance AIfES with more advanced matrix multiplication methods for ANNs and optimize the overall on-device training for ANN with e.g. pruning. Furthermore, new ANN architectures, such as transformers will be added, and we will focus on the further development of FL and OL techniques with AIfES.

REFERENCES

[1] M. Frank, D. Drikakis, and V. Charissis, "Machine-learning methods for computational science and engineering," *Computation*, vol. 8, no. 1, pp. 15, 2020, doi: 10.3390/computation8010015.
[2] R. Cioffi, M. Travaglioni, G. Piscitelli, A. Petrillo, and F. de Felice, "Artificial intelligence and machine learning applications in smart production: Progress, trends, and directions," *Sustainability*, vol. 12, no. 2, 2020, Art. no. 492.
[3] H. Ren, D. Anicic, and T. A. Runkler, "TinyReptile: TinyML with federated meta-learning," in *Proc. Int. Joint Conf. Neural Netw.*, 2023, pp. 1–9.
[4] 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, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157821003335
[5] A. Mostafavi and A. Sadighi, "A novel online machine learning approach for real-time condition monitoring of rotating machines," in *Proc. 9th RSI Int. Conf. Robot. Mechatronics*, 2021, pp. 267–273.
[6] D. Pau, A. Khiari, and D. Denaro, "Online learning on tiny microcontrollers for anomaly detection in water distribution systems," in *Proc. IEEE 11th Int. Conf. Consum. Electron.*, 2021, pp. 1–6.
[7] K. Sai Charan, "An auto-encoder based TinyML approach for real-time anomaly detection," in *Proc. 10th SAE India Int. Mobility Conf.*, 2022, pp. 2022–28-0406.

[8] T. Kohlheb, M. Sinapius, C. Pommer, and A. Boschmann, "Embedded autoencoder-based condition monitoring of rotating machinery," in *Proc. IEEE 26th Int. Conf. Emerg. Technol. Factory Automat.*, 2021, pp. 1–4.

[9] H. Bosman, A. Liotta, G. Iacca, and H. Wörtche, "Anomaly detection in sensor systems using lightweight machine learning," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, 2013, pp. 7–13.

[10] C. Antonopoulos, A. Prayati, T. Stoyanova, C. Koulamas, and G. Papadopoulos, "Experimental evaluation of a WSN platform power consumption," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–8.

[11] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[12] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[13] R. David et al., "TensorFlow lite micro: Embedded machine learning on TinyML systems," 2020, *arXiv: 2010.08678*. [Online]. Available: https://arxiv.org/abs/2010.08678

[14] T. Chen et al., "TVM: End-to-end optimization stack for deep learning," vol. 11, 2018, *arXiv: 1802.04799*.

[15] D. Nadalini, M. Rusci, G. Tagliavini, L. Ravaglia, L. Benini, and F. Conti, "PULP-trainlib: Enabling on-device training for RISC-V multi-core MCUs through performance-driven autotuning," in *Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Modeling, Simul.*, A. Orailoglu, M. Reichenbach, and M. Jung, Eds., Cham, Springer International Publishing, 2022, pp. 200–216.

[16] F. Sakr, R. Berta, J. Doyle, A. De Gloria, and F. Bellotti, "Self-learning pipeline for low-energy resource-constrained devices," *Energies*, vol. 14, no. 20, pp. 1–19, 2021. [Online]. Available: https://www.mdpi.com/1996--1073/14/20/6636

[17] N. L. Giménez, F. Freitag, J. Lee, and H. Vandierendonck, "Comparison of two microcontroller boards for on-device model training in a keyword spotting task," in *Proc. 11th Mediterranean Conf. Embedded Comput.*, 2022, pp. 1–4.

[18] H. Ren, D. Anicic, and T. A. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," in *Proc. Int. Joint Conf. Neural Netw.*, 2021, pp. 1–8.

[19] S. Lee, B. Islam, Y. Luo, and S. Nirjon, "Intermittent learning: On-device machine learning on intermittently powered system," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 3, no. 4, pp. 1–30, Dec. 2019.

[20] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256KB memory," in *Proc. Adv. Neural Inf. Process. Syst.*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., Curran Associates, Inc., vol. 35, 2022, pp. 22941–22954. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/90c56c77c6df45fc8e556a096b7a2b2e-Paper-Conference.pdf

[21] B. Sudharsan, J. G. Breslin, and M. I. Ali, "Edge2Train: A framework to train machine learning models (SVMs) on resource-constrained IoT edge devices," in *Proc. 10th Int. Conf. Internet Things*, New York, NY, USA, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3410992.3411014

[22] G. Delnevo, S. Mirri, C. Prandi, and P. Manzoni, "An evaluation methodology to determine the actual limitations of a tinyml-based solution," *Internet Things*, vol. 22, 2023, Art. no. 100729. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660523000525

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[24] B. Sudharsan, P. Yadav, J. G. Breslin, and M. Intizar Ali, "Train : An incremental ML model training algorithm to create self-learning IoT devices," in *Proc. IEEE SmartWorld, Ubiquitous Intell. Comput., Adv. Trusted Comput., Scalable Comput. Commun., Internet People Smart City Innov.*, 2021, pp. 97–106.

[25] I. Hoyer, A. Utz, A. Lüdecke, M. Rohr, C. H. Antink, and K. Seidl, "Inference runtime of a neural network to detect atrial fibrillation on customized RISC-V-based hardware," *Curr. Directions Biomed. Eng.*, vol. 8, no. 2, pp. 703–706, 2022. [Online]. Available: https://doi.org/10.1515/cdbme-2022--1179

[26] A. Moss, H. Lee, L. Xun, C. Min, F. Kawsar, and A. Montanari, "Ultra-low power DNN accelerators for IoT: Resource characterization of the max78000," in *Proc. 20th ACM Conf. Embedded Netw. Sensor Syst.*, New York, NY, USA, Association for Computing Machinery, 2023, pp. 934–940. [Online]. Available: https://doi.org/10.1145/3560905.3568300

[27] D.-M. Ngo et al., "HH-NIDS: Heterogeneous hardware-based network intrusion detection framework for IoT security," *Future Internet*, vol. 15, no. 1, pp. 1–20, 2023. [Online]. Available: https://www.mdpi.com/1999--5903/15/1/9

[28] L. Wulfert, C. Wiede, and A. Grabmaier, "TinyFL: On-device training, communication and aggregation on a microcontroller for federated learning," in *Proc. IEEE 21st Interregional NEWCAS Conf.*, 2023, pp. 1–5.

[29] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli, "TinyML platforms benchmarking," 2021, *arXiv:2112.01319*.

[30] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[31] EdgeImpuls, "Edgeimpuls," 2022, Accessed: Oct. 10, 2022. [Online]. Available: https://www.edgeimpulse.com/

[32] STMicroelectronics, "STM32cube.AI," 2022, Accessed: May 17, 2022. [Online]. Available: https://www.st.com/content/st_com/en/ecosystems/artificial-intelligence-ecosystem-stm32.html

[33] STMicroelectronics, "X-cube-AI," 2023, Accessed: Oct. 11, 2023. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html

[34] F. Chollet et al., "Keras," 2015. [Online]. Available: https://keras.io

[35] O. R. Developers, "ONNX runtime," 2021. [Online]. Available: https://onnxruntime.ai/

[36] Cartesiam, "Nanoedge AI studio," 2022, Accessed: Oct. 10, 2022. [Online]. Available: https://cartesiam-neai-docs.readthedocs-hosted.com/index.html

[37] Microsoft, "Embedded learning library," 2020, Accessed: Nov. 15, 2022. [Online]. Available: https://github.com/microsoft/ELL

[38] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. 4th Int. Conf. Learn. Representations*, San Juan, Puerto Rico, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[39] Y. He et al., "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. 15th Eur. Conf. Comput. Vis.*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., Cham, Springer International Publishing, 2018, pp. 815–832.

[40] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 1398–1406.

[41] E. Liberis and N. D. Lane, "Differentiable neural network pruning to enable smart applications on microcontrollers," vol. 6, no. 4, Jan.pp. 1–19, 2023. [Online]. Available: https://doi.org/10.1145/3569468

[42] J. Lin et al., "Runtime neural pruning," in *Proc. Adv. Neural Inf. Process. Syst.*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 2181–2191. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/a51fb975227d6640e4fe47854476d133-Paper.pdf

[43] J. Liu et al., "Discrimination-aware network pruning for deep model compression," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 8, pp. 4035–4051, Aug. 2022.

[44] X. Ding et al., "ResRep: Lossless CNN pruning via decoupling remembering and orgetting," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2021, pp. 4490–4500.

[45] J. Choi et al., "PACT: Parameterized clipping activation for quantized neural networks," 2018, *arXiv: 1805.06085*. [Online]. Available: https://api.semanticscholar.org/CorpusID:21721698

[46] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Cham, Springer International Publishing, 2016, pp. 525–542.

[47] F. Daghero et al., "Human activity recognition on microcontrollers with quantized and adaptive deep neural networks," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 4, pp. 1–28, Aug. 2022. [Online]. Available: https://doi.org/10.1145/3542819

[48] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers," 2019, *arXiv: 1905.13082*. [Online]. Available: http://arxiv.org/abs/1905.13082

[49] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 8612–8620.

[50] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 2704–2713, doi: 10.1109/CVPR.2018.00286.

[51] W. Chen et al., "Quantization of deep neural networks for accurate edge computing," *J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 4, pp. 54:1–54:11, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3451211

[52] 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.*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H.Lin, Eds., Curran Associates, Inc., 2020, pp. 11711–11722. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/86c51678350f656dcc7f490a43946ee5-Paper.pdf

[53] S. Xu, H.B. LiJ. ZhuangJ. LiuC. CaoLiang, and M. Tan, "Generative low-bitwidth data free quantization," in *Proc. 16th Eur. Conf. Comput. Vis.*, A.H. VedaldiBischof, T. Brox, and J.-M. Frahm, Eds., Cham, Springer International Publishing, 2020, pp. 1–17.

[54] Z. Xie, Z. Wen, J. Liu, Z. Liu, X. Wu, and M. Tan, "Deep transferring quantization," in *Proc. Eur. Conf. Comput. Vis.*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., Cham, Springer International Publishing, 2020, pp. 625–642.

[55] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "MCUNetV2: Memory-efficient patch-based inference for tiny deep learning," 2021, *arXiv:2110.15352*, doi: 10.48550/arXiv.2110.15352.

[56] Z. Sun et al., "Entropy-driven mixed-precision quantization for deep network design," in *Proc. Ad. Neural Inf. Process. Syst.*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., Curran Associates, Inc., 2022, pp. 21508–21520. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/86e7ebb16d33d59e62d1b0a079ea058d-Paper-Conference.pdf

[57] B. Zoph, V. Vasudevan, J. Shlens, and Q. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 8697–8710.

[58] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, 2017. [Online]. Available: https://openreview.net/forum?id=r1Ue8Hcxg

[59] Y. Geifman and R. El-Yaniv, "Deep active learning with a neural architecture search," in *Proc. Adv. Neural Inf. Process. Syst.*, H.H. WallachA. LarochelleF. Beygelzimer d E. Alché-BucFox, and R.Garnett, Eds., Curran Associates, Inc., 2019, pp. 5974–5984. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/b59307fdacf7b2db12ec4bd5ca1caba8-Paper.pdf

[60] C. LiP. HaoLi, J. Xiong, and D. Chen, "Generic neural architecture search via regression," in *Proc. Adv. Neural Inf. Process. Syst.*, M.A. RanzatoY. BeygelzimerDauphin, P.Liang, and J. W. Vaughan, Eds., Curran Associates, Inc., 2021, pp. 20476–20490. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/aba53da2f6340a8b89dc96d09d0d0430-Paper.pdf

[61] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, "A comprehensive survey on hardware-aware neural architecture search," 2021. [Online]. Available: https://arxiv.org/abs/2101.09336

[62] Y. Guo et al., "Towards accurate and compact architectures via neural architecture transformer," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 10, pp. 6501–6516, Oct. 2022.

[63] F. De Vita, G. Nocera, D. Bruneo, V. Tomaselli, and M. Falchetto, "On-device training of deep learning models on edge microcontrollers," in *Proc. IEEE Int. Conf. Internet Things IEEE Green Comput. Commun. IEEE Cyber, Phys. Soc. Comput. IEEE Smart Data IEEE Congr. Cybermatics*, 2022, pp. 62–69.

[64] B. Sudharsan, J. G. Breslin, and M. I. Ali, "ML-MCU: A framework to train ML classifiers on MCU-based IoT edge devices," *IEEE Internet Things J.*, vol. 9, no. 16, pp. 15007–15017, Aug. 2022.

[65] F. De Vita, R. M. A. Nawaiseh, D. Bruneo, V. Tomaselli, M. Lattuada, and M. Falchetto, "-ff: On-device forward-forward training algorithm for microcontrollers," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, 2023, pp. 49–56.

[66] N. Reddy KP, Y. Geyavalli, D. Sujani, and M. S. Rajesh, "Comparison of programming languages: Review," *Int. J. Comput. Sci. Commun.*, vol. 9, pp. 113–122, Jul. 2018.

[67] T. FunkBucksch and D. Mueller-Gritschneder, "ML training on a tiny microcontroller for a self-adaptive neural network-based dc motor speed controller," in *Proc. Int. Workshop IoT Streams Data-Driven Predictive Maintenance IoT, Edge, Mobile Embedded Mach. Learn.*, J.S. GamaA. PashamiM. BifetH. Sayed-MouchaweF. FröningPernkopf, G. Schiele, and M. Blott, Eds., Cham, Springer International Publishing, 2020, pp. 268–279.

[68] A. Mostafavi and A. Sadighi, "A novel online machine learning approach for real-time condition monitoring of rotating machines," in *Proc. 9th RSI Int. Conf. Robot. Mechatronics*, 2021, pp. 267–273.

[69] J. Guan and G. Liang, "A research of convolutional neural network model deployment in low- to medium-performance microcontrollers," in *Proc. 10th Int. Conf. Wireless Commun. Sensor Netw.*, New York, NY, USA, 2023, pp. 44–50. [Online]. Available: https://doi.org/10.1145/3585967.3585975

[70] R. Heymsfeld, Arduinoann. 2022, Accessed: Oct. 10, 2022. [Online]. Available: http://robotics.hobbizine.com/arduinoann.html

[71] K. Kopparapu, E. Lin, J. G. Breslin, and B. Sudharsan, "TinyFedTL: Federated transfer learning on ubiquitous tiny IoT devices," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops other Affiliated Events*, 2022, pp. 79–81.

[72] N. Llisterri Giménez, J. Miquel Solé, and F. Freitag, "Embedded federated learning over a LoRa mesh network," *Pervasive Mobile Comput.*, vol. 93, 2023, Art. no. 101819. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574119223000779

[73] N. Llisterri Giménez, M. Monfort Grau, R. Pueyo Centelles, and F. Freitag, "On-device training of machine learning models on microcontrollers with federated learning," *Electronics*, vol. 11, no. 4, 2022, Art. no. 573.

[74] G. Hinton, "The forward-forward algorithm: Some preliminary investigations," 2022, *arXiv:2212.13345*, doi: 10.48550/arXiv.2212.13345.

[75] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note'," German National Research Center for Information Technology, Bonn, Germany, GMD Tech. Rep. 148, 2001.

[76] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Comput.*, vol. 11, no. 4, pp. 853–862, 1999.

[77] D. Costa, M. Costa, and S. Pinto, "Train me if you can: Decentralized learning on the deep edge," *Appl. Sci.*, vol. 12, no. 9, 2022. [Online]. Available: https://www.mdpi.com/2076--3417/12/9/4653

[78] N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computat.*, vol. 11, pp. 853–62, Jun. 1999, doi: 10.1162/089976699300016467.

[79] PlatformIO, "PlatformIO core," 2022. [Online]. Available: https://platformio.org/

[80] ARM. GNU arm embedded toolchain, 2021. Accessed: Nov. 15, 2022. [Online]. Available: https://developer.arm.com/downloads/-/gnu-rm

[81] TensorFlow, "TensorFlow lite for microcontrollers," 2022. [Online]. Available: https://github.com/tensorflow/tflite-micro

[82] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm cortex-M CPUs," 2018, *arXiv: 1801.06601*.

[83] B. Sudharsan et al., "TinyML benchmark: Executing fully connected neural networks on commodity microcontrollers," in *Proc. IEEE 7th World Forum Internet Things*, 2021, pp. 883–884.

[84] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," [Online]. Available: http://yann.lecun.com/exdb/mnist/

[85] R. Fisher, "Iris," 1988. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/iris

[86] W. Wolberg, W. Street, and O. Mangasarian, "Breast cancer wisconsin (diagnostic)," 1995. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Breast_Cancer_Wisconsin%28Diagnostic%29

[87] A. Krizhevsky, Learning multiple layers of features from tiny images. 2009, Accessed: Nov. 17, 2022. [Online]. Available: https://www.cs.toronto.edu/kriz/learning-features-2009-TR.pdf

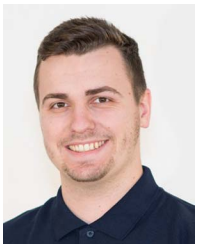
[88] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes, "Visual wake words dataset," 2019, *arXiv: 1906.05721*. [Online]. Available: http://arxiv.org/abs/1906.05721

[89] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn.*, 2015, pp. 448–456.

[90] C. R. Banbury et al., "MLPerf tiny benchmark," 2021, *arXiv:2106.07597*. [Online]. Available: https://arxiv.org/abs/2106.07597

Lars Wulfert received the BSc degree in physical engineering from the University of Applied Sciences Gelsenkirchen, Gelsenkirchen, Germany, in 2018, and the MSc degree in microsystem technology from the Applied Sciences Gelsenkirchen, Gelsenkirchen, Germany, in 2020. He is currently working toward the PhD degree with the Fraunhofer Institute for Microelectronic Circuits and Systems. His research interests include federated learning and embedded systems.

Johannes Kühnel received the BEng degree in electrical engineering from the University of Applied Sciences Bielefeld, Bielefeld, Germany, in 2019 and the MSc degree in embedded systems engineering from the University of Duisburg-Essen, Duisburg, Germany, in 2021. He is currently working toward the PhD degree with the Fraunhofer Institute for Microelectronic Circuits and Systems. His research interests include machine learning, embedded systems, and signal analysis.



Lukas Krupp received the BSc degree in electrical and computer engineering from the Technical University of Kaiserslautern, Kaiserslautern, Germany, in 2018, and the MSc degree in electrical and computer engineering from the Technical University of Kaiserslautern, Kaiserslautern, Germany, in 2019. He is currently working toward the PhD degree with the Fraunhofer Institute for Microelectronic Circuits and Systems. His research interests include machine learning, predictive maintenance, and embedded systems.

Justus Viga received the BSc degree in electrical engineering, information technology and computer engineering from RWTH Aachen University, Aachen, Germany, in 2019. He is currently working towad the MSc degree in computer engineering with RWTH Aachen University. From 2019 to 2022, he worked on tiny-ML projects with the Fraunhofer Institute for Microelectronic Circuits and Systems as a student assistant. His research interests are mainly focused on the field of AI in embedded applications.

Christian Wiede received the BSc and MSc degrees in biomedical engineering from the Ilmenau University of Technology, Ilmenau, Germany, in 2011 and 2013, respectively, and the PhD degree in electrical engineering and information technology from the Chemnitz University of Technology, Chemnitz, Germany, in 2018. He is specialized in the field of computer vision, machine learning, artificial intelligence and their applications in industry an medicine. He is currently working with Fraunhofer IMS as head of Embedded AI.

Pierre Gembaczka received the MSc degree in microtechnology and medical engineering from the University of Applied Sciences Gelsenkirchen, Gelsenkirchen, Germany, in 2010, and the PhD degree in electrical engineering from the University of Duisburg-Essen, Duisburg, Germany, in 2014. Since 2010, he has been with the Fraunhofer Institute for Microelectronic Circuits and Systems, first receiving the PhD degree, where he is currently program manager for the "Industrial AI" topic and product manager for the AI software framework AIfES.

Anton Grabmaier received the MS and PhD degrees in physics from the University of Stuttgart, Stuttgart, Germany, in 1989 and 1993, respectively. He was a post-doctoral with the Deutsche Telekom Research Center, where he was engaged in modeling, technology, and characterization of DFB lasers. From 1994 to 1999, he has been with Valeo Switches and Sensors GmbH where he worked on the development of advanced electrical systems in automotives. From 1999 to 2005, he has been a director in Siemens VDO (Sensor Innovation Management) responsible for the worldwide production of new sensors. In 2006, he joined the University of Duisburg Essen, Duisburg, Germany, as a full university professor with the Electrical Engineering Faculty, where he is currently chair of the Electronic Components and Circuits Department. Since 2006, he is the head with the Fraunhofer Institute IMS, Duisburg. He has published more than 50 technical papers and articles. Also, he is currently a member of the German Chamber of Industry and Commerce for Research and Technology, the Eduard Rhein Foundation, Program Committee of the Microsystems Technology Congress of the GMM VDE/VDI Society and University Council of Hochschule Ruhr West. In 2009, he received the Honorary Membership in the German Research Hall of Fame.