sensors

MDPI

Article

Quantization and Deployment of Deep Neural Networks on Microcontrollers

Pierre-Emmanuel Novac [1,*], Ghouthi Boukli Hacene [2,3], Alain Pegatoquet [1], Benoît Miramond [1] and Vincent Gripon [2]

[1] CNRS, LEAT, Université Côte d'Azur, 06903 Sophia Antipolis, France; alain.pegatoquet@univ-cotedazur.fr (A.P.); Benoit.Miramond@univ-cotedazur.fr (B.M.)
[2] IMT Atlantique, 29200 Brest, France; Ghouthi.BouckliHacene@imt-atlantique.fr (G.B.H.); vincent.gripon@imt-atlantique.fr (V.G.)
[3] MILA, Montreal, QC H2S 3H1, Canada
* Correspondence: Pierre-Emmanuel.Novac@univ-cotedazur.fr

Abstract: Embedding Artificial Intelligence onto low-power devices is a challenging task that has been partly overcome with recent advances in machine learning and hardware design. Presently, deep neural networks can be deployed on embedded targets to perform different tasks such as speech recognition, object detection or Human Activity Recognition. However, there is still room for optimization of deep neural networks onto embedded devices. These optimizations mainly address power consumption, memory and real-time constraints, but also an easier deployment at the edge. Moreover, there is still a need for a better understanding of what can be achieved for different use cases. This work focuses on quantization and deployment of deep neural networks onto low-power 32-bit microcontrollers. The quantization methods, relevant in the context of an embedded execution onto a microcontroller, are first outlined. Then, a new framework for end-to-end deep neural networks training, quantization and deployment is presented. This framework, called MicroAI, is designed as an alternative to existing inference engines (TensorFlow Lite for Microcontrollers and STM32Cube.AI). Our framework can indeed be easily adjusted and/or extended for specific use cases. Execution using single precision 32-bit floating-point as well as fixed-point on 8- and 16 bits integers are supported. The proposed quantization method is evaluated with three different datasets (UCI-HAR, Spoken MNIST and GTSRB). Finally, a comparison study between MicroAI and both existing embedded inference engines is provided in terms of memory and power efficiency. On-device evaluation is done using ARM Cortex-M4F-based microcontrollers (Ambiq Apollo3 and STM32L452RE).

Keywords: embedded systems; artificial intelligence; machine learning; quantization; power consumption; microcontrollers

Citation: Novac, P.-E.; Boukli Hacene, G.; Pegatoquet, A.; Miramond B.; Gripon V. Quantization and Deployment of Deep Neural Networks on Microcontrollers. *Sensors* **2021**, *21*, 2984. https://doi.org/10.3390/s21092984

Academic Editor: Alexander Wong

Received: 29 March 2021
Accepted: 20 April 2021
Published: 23 April 2021

1. Introduction

Deep Neural Networks (DNN) are presently a widespread tool to solve a wide range of problems including classification. DNN can indeed classify all sorts of data such as audio, images or accelerometer samples for tasks such as speech recognition, object recognition or Human Activity Recognition (HAR).

A well-known downside of DNN is its high energy consumption requirement. Indeed, the training phase is usually based on a large amount of data processed by costly algorithms. Although the inference phase requires less processing power, it is still a costly process. GPUs and ASICs are then often used in the Cloud to perform computations of such tasks [1].

However, Cloud computing requires transmitting the collected data to a network server to process it and fetch the result, thus requiring a permanent connectivity, causing privacy concerns as well as non-deterministic latency. As an alternative, computations can be done at the edge on the device itself. By doing so, data do not need to be sent by the device to the cloud anymore. However, running DNN on resource-constrained

devices such as a microcontroller used in Internet of Things (IoT) devices or wearables is a challenging task [2–4].

These devices have indeed only a very small amount of memory, often less than 1 MiB. They also run DNN algorithms several orders of magnitude slower than GPUs or even CPUs (see Appendix A). The reason is that microcontrollers generally rely on a general-purpose processing core that does not implement parallelism techniques such as thread-level parallelism or advanced vectorization. Moreover, microcontrollers typically run at a much lower frequency than GPU (8 MHz to 80 MHz compared to 1 GHz to 2 GHz). Microcontrollers can also be coupled with tiny battery cells. In some cases, for example when data are collected in remote areas, they cannot even be recharged in the field. Therefore, performing inference at the edge faces major issues in terms of real-time constraint, power consumption and memory footprint. To meet these embedded constraints, the deployment of a DNN must respect an upper bound for one inference response time as well as an upper bound for the number of parameters of the network.

As a result, a DNN must be limited in width and depth to be deployed on a microcontroller. As it has been observed many times so far, deeper and/or wider networks are often able to solve more complex tasks with better accuracy [5]. As such, there is always a trade-off between memory footprint, response time, power consumption and accuracy of the model. In a previous work [6], we already presented a trade-off between memory footprint, power consumption and accuracy when performing HAR onto smart glasses. This work has shown that HAR is feasible in real time on a low-power Cortex-M4F-based microcontroller. However, we also concluded that there was room for memory footprint and power consumption improvement.

A technique that can provide a significant decrease of the memory footprint is based on network quantization. Quantization consists of reducing the number of bits used to encode each weight of the model, so that the total memory footprint is reduced by the same factor. Quantization also enables the use of fixed-point rather than floating-point encoding. In other words, operations can be performed using integer rather than floating-point data types. This is something interesting as integer operations require much less computations on most processor cores, including microcontrollers. Without a Floating-Point Unit (FPU), floating-point instructions must be emulated in software, thus creating a large overhead as it has been illustrated in [7]. In this study, a comparison between software, hardware and hybrid custom FPU implementation is provided.

In this paper, we present a framework, called MicroAI, to perform end-to-end training, quantization and deployment of deep neural networks on microcontrollers. The training phase relies on well-known TensorFlow and PyTorch deep learning frameworks. Our objective is to provide a framework easy to adapt and extend, while maintaining a good compromise between accuracy, energy efficiency and memory footprint.

As a second contribution we provide some comparative results using two different microcontrollers (STM32L452RE and Ambiq Apollo3) and three different inference engines (TensorFlow Lite for Microcontrollers, STM32Cube.AI and our own MicroAI). Results are compared in terms of memory footprint, inference time and power efficiency. Finally, we propose to apply 8-bit and 16-bit quantization methods on three datasets dealing with different modalities: acceleration and angular velocity from body-worn sensors for UCI-HAR, speech for Spoken MNIST and images for GTSRB. These datasets are light enough to be handled by a deep neural network running on a microcontroller, but still relevant for applications relying on embedded artificial intelligence.

Section 2 presents the challenges of running deep neural networks on microcontrollers and recent advances in this field. Section 3 describes the two common ways of representing real numbers on modern computers: floating-point and fixed-point. Section 4 presents the methodology implemented in our MicroAI framework for deep neural networks quantization. Section 5 details our MicroAI framework and compares it to existing solutions. In Section 6, some comparative results between our framework MicroAI and two popular embedded neural network frameworks (TensorFlow Lite for Microcontrollers

and STM32Cube.AI) are given for two microcontroller platforms (Nucleo-L452RE-P and SparkFun Edge) in terms of inference time and power efficiency. Impact of our 8- and 16-bit quantization methods for three different datasets (UCI-HAR, Spoken MNIST and GTSRB) is also presented. In Section 7 a discussion on obtained results is given. Finally, Section 8 concludes this work and discusses future perspectives.

2. State of the Art on Embedded Execution of Quantized Neural Networks

A first family of optimization methods for embedded deep neural networks, based on network quantization, proposes to reduce the precision (i.e., the number of bits used to represent a value). Numerous works propose to use low-bit quantization (i.e., 2 or 3 bits to quantize both weights and activations) such as PArameterized Clipping acTivation (PACT) combined to Statistics-Aware Weight Binning (SAWB) [8], Learned Step Size Quantization (LSQ) [9], Bit-Pruning [10] or Differentiable Quantization of Deep Neural Networks [11] (DQDNN). In the extreme case, this amounts to binarizing the network parameters [12,13]. Nevertheless, it is usually possible to find a good compromise between the precision and the performance of a given architecture.

Another family of methods focuses on low-cost architectures. This is notably the case of the well-known MobileNet [14] and Squeeze-Net [15] networks. Finding a good architecture for a given problem is difficult, and remains an open issue even if hardware constraints are not taken into account. Indeed, the combinatorial explosion of hyperparameters makes the exploration of an architecture very expensive. These networks are often tailored to solve computer vision problems such as ImageNet [16] classification, and are therefore not well suited for general use cases or simpler problems. In practice, many applications use a generic network such as ResNet [17]. This is also the solution retained in this work. The reason is that we want to easily make use of the same deep neural network on different kind of data (time series, audio spectrum and image) and simplify the implementation.

In a third family of possibilities, some authors explored the possibility of reducing the number of parameters in neural networks by identifying parts of the network that are not very useful for decision making. These parts can then be removed from the architecture. These pruning techniques make it possible to considerably reduce the number of parameters. Hence, in [18], the authors manage to remove up to 90% of the number of parameters. However, the unstructured nature of the removed parameters makes it difficult to operate the network. Recently, several works have been proposed to prune in a structured way where a whole kernel, a filter of even a layer is pruned according to a specific criterion [19–22].

Finally, a last family of methods consists of identifying similar parts at various points in the architectures to factorize them. For example, in [23] the authors demonstrate that it is possible to reduce the number of parameters by replacing them with memory pointers. Some works propose to improve accuracy by adding some learning steps and algorithms to get a more representative factorization [24,25].

All these compression techniques make it possible to considerably reduce the size of architectures, but usually at the cost of a reduction in performance. In practice, it is also common to reduce by two or even two thirds the memory used to store the parameters while maintaining a similar level of performance.

In this work we will focus on quantization-based compression. To reduce the number of bits used to represent a value, quantization maps values from one set to another smaller set. The smaller set can have a constant step between its elements, in which case the quantization scheme is said to be uniform. In the case of convolutional neural networks, the authors in [26] show that the weights of convolutional layers typically follow a Gaussian distribution when weight decay is applied. More generally, it has been shown that weights can closely fit Gaussian Mixture Models [27]. Therefore, choosing a non-uniform quantization scheme to better represent the values of the non-unform distribution of weights would allow for a lower quantization error.

A non-uniform quantization scheme was implemented in [28] on an FPGA device. In this work, instead of coding the value in a fixed-point format, only the nearest power of two is coded. Using this approach, it is possible to obtain a better resolution compared to a fixed-point representation for numbers near 0. This approach also allows large values to be represented, but at the cost of a lower resolution. The quantization step is determined by minimizing the quantization error at the output of the layer, thus balancing the precision and the dynamic range. As the implementation relies on bit shifts rather than on integer multiplications, this solution has some benefits in terms of resource usage and latency for an FPGA target. Additionally, results show that there is a slight accuracy degradation when using the proposed non-uniform quantization versus a uniform quantization.

Using lower precision computation for deep neural networks has already been explored in [29] where the authors compare the test error rates on various image datasets for single precision floating-point, half-precision floating-point, 20-bit fixed-point and their own dynamic fixed-point approach with 10 bits for activations and 12 bits for weights. In their work, it is worth noting that the training is also performed using a lower precision arithmetic. Training with fixed-point arithmetic was already presented in [30] with 16-bit weights and 8-bit inputs, causing an accuracy loss of a few percent in their evaluation of text-to-speech, parity bit computation, protein structure prediction and sonar signals classification problems. In [31], the authors show that on an Intel® E5640 microprocessor with an x86 architecture, using 8-bit integer instructions instead of floating-point instructions provides an execution speedup of more than 2, without a loss of accuracy on a speech recognition problem. In this case, the training is performed using single precision floating-point arithmetic, and the evaluation is done after the quantization of the network parameters.

These prior works were however mostly out of the scope of embedded computing on microcontrollers. Running deep neural networks on microcontrollers started to be popular in the last few years thanks to the rise of the Internet of Things and the improved efficiency of deep neural networks.

In [32], the authors insist on the fact that the ISA (Instruction Set Architecture) of available microcontrollers can be a great limitation to running quantized neural networks. Indeed, most microcontroller architectures do not exhibit any kind of SIMD instructions. On the other hand, most of these microcontrollers rely on 32-bit registers. Thus, even if the neural network parameters and the input data use a lower precision representation, they must be computed one by one using the register width of 32 bits. Some more advanced microcontroller architectures offer instructions able to handle 4 × 8-bit or 2 × 16-bit data packed in 32-bit registers. However, such advances do not allow working with intermediate precision or sub-byte precision, and not all arithmetic and logic instructions are covered. Intermediate or sub-byte precision requires manually packing and unpacking data, thus inducing a noticeable computation overhead, even though it helps further reducing the memory footprint.

Moreover, microcontrollers used in IoT mostly rely on ARM Cortex-M cores with the associated ISA. However, ARM cores are not open, meaning that modifying and extending the ISA is not possible. To overcome these limitations, the authors in [32] proposed an extension to the RISC-V ISA, which is open, with instructions to handle sub-byte quantization. Unfortunately, as microcontrollers implementing RISC-V are still scarce on the market, and not readily available with the proposed extension, this approach cannot be reasonably used to deploy IoT devices since it requires manufacturing a custom microcontroller. Manufacturing a custom microcontroller is not feasible when the goal is to release an IoT product on the market due to large costs, time and the required level of expertise. As a result, only off-the-shelf microcontrollers are considered in this work. Only 8-bit, 16-bit and 32-bit precision will therefore be studied.

Deep neural networks have already been deployed on 8-bit microcontrollers. One of the first method was proposed in [33]. Although interesting, this method requires a lot of work to implement pseudo-floating-point coding, a custom multiplication algorithm over 16 bits, as well as a hyperbolic tangent approximation for the activation function,

all in assembly language. Over the last few years, implementations have relied on 32-bit microcontrollers with either a hardware FPU or fixed-point computations. Additionally, the Rectified Linear Unit (ReLU) [34] became widely used as an activation function and has the benefit of being easily computed as a max between 0 and the layer's output, thus being much less complex than a hyperbolic tangent. In the meantime, neural network architectures and training methods continued to evolve to provide more and more efficient models. As a result, applications such as spoken keyword spotting [35] and Human Activity Recognition [6] can now be performed in real time on IoT devices relying on low-power microcontrollers.

3. Real Numbers Representation

3.1. Floating-Point

In modern computation systems, real numbers typically use a floating-point representation. A floating-point representation relies on the encoding of three different information: the sign, the significand and the exponent. Coding the significand and the exponent separately allows representing values with a very large dynamic range, while at the same time providing increasing precision as the numbers approach 0.

Most floating-point implementations follow the IEEE754 [36] standard which defines how the sign, significand and exponent are coded in a binary format. Floating-point numbers can be coded in half, single or double precision requiring 16-, 32- or 64-bit, respectively. Obviously, the more bits allocated to code a value, the more precise it is. A higher number of bits allocated to the exponent also allows for a larger dynamic range. In deep neural networks, single precision is more than enough for training and inference. Double precision requires more computing resources, so it is generally not used. Recently, it has been shown that half-precision can further accelerate the training and inference without a significant drop in the accuracy of the deep neural networks [37].

However, the choice is much more restricted for low-power microcontrollers. When present, the hardware floating-point unit often only supports single precision computation. The double precision must be computed in software and is therefore significantly slower than single precision. The half-precision is converted to single precision before the computation. In 2019, ARM released the ARMv8.1-M ISA which includes instructions for half-precision support. Even though the Cortex-M55 core is planned to implement these instructions, there is so far no microcontroller with this core available on the market. As a result, when floating-point is used on a microcontroller, only single precision is considered.

The binary representation of single precision floating-point numbers is called binary32 and is represented with 1 bit for the sign, 8 bits for the exponent and 23 bits for the significand (see Table 1). It allows a dynamic range of roughly $[-10^{38}, 10^{38}]$, far beyond the values typically seen in a deep neural network, while increasing the resolution for numbers close to 0. The number closest to 0 are about $\pm 1.4 \times 10^{-45}$.

Table 1. Single precision floating-point binary32 representation.

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

3.2. Fixed-Point

Fixed-point is another way to represent real numbers. In this representation, the integer part and the fractional part have a fixed length. As a result, the dynamic range and the resolution are directly limited by the length of the integer part and the length of the fractional part, respectively. The resolution is constant across the whole dynamic range. In binary, the Q notation is often used to specify the number of bits associated with each part. $Qm.n$ is a number where m bits are allocated to the integer part and n bits to the fractional part [38]. It is important to note that we consider signed numbers in two's complement representation, the sign bit being included in the integer part. The number of bits

for the integer part can be increased to obtain a larger dynamic range, but it will conversely reduce the number of bits allocated to the fractional part, thus reducing its precision.

Given a $Qm.n$ signed number, its dynamic range is $[-2^{m-1}, 2^{m-1} - 2^{-n}]$ and its resolution is 2^{-n}.

As an example, in Table 2, a signed $Q16.16$ number stored into a 32-bit register has 16 bits for the integer part including 1 bit for the sign and 16 bits for the fractional part. This translates to a dynamic range of $[-32,768, 32,767.9999847]$, much smaller than the equivalent floating-point representation, and a constant resolution of 1.5259×10^{-5} across the whole range, less precise than the floating-point representation near 0.

Table 2. Fixed-point Q16.16 on 32-bit representation.

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

4. Training and Quantization of Deep Neural Networks

In this work, the training is always performed using single precision floating-point computation, i.e., in the *binary32* format. As training is done offline on a workstation, there is no need to perform it in fixed-point. Despite it being feasible, it would come with additional challenges regarding gradient computation.

4.1. Floating-Point to Fixed-Point Quantization of a Deep Neural Network

Since the training relies on floating-point computation, a conversion from a floating-point to a fixed-point representation must be performed before the deep neural network is deployed on the target. As the set of possible values is different between floating-point and fixed-point representations, this involves quantizing the weights of the deep neural network.

Floating-point to fixed-point conversion requires determining a scale factor, so that the floating-point number can be represented as an integer multiplied by a scale factor. The scale factor is a positive or negative power of two so that it can be computed using only left or right shifts. In the case of the Cortex-M4 architecture, both multiplication and shift instructions take one cycle. However, a division requires 2 to 12 cycles. Therefore, divisions should be avoided as much as possible.

The scale factor must be chosen to represent the whole range of values while avoiding any risk of data overflow, but at the cost of a lower precision for smaller numbers.

4.1.1. Uniform and Non-Uniform

Similarly to the works presented in Section 2, in our experiments, we also observed that convolutional layer weights are close to Gaussian distributions, with a mean close to 0 when inputs are normalized. Such a distribution of weights for a convolutional layer kernel is shown in Figure 1. As a result, convolutional layer weights can be better represented using a non-uniform distribution of numbers. This is what floating-point numbers originally do to get a better precision around 0.

However, as the goal is to perform fast computations, a uniform quantization is preferred. Non-uniform quantization would indeed require performing additional transformations before using the microcontroller's instructions. This overhead can be non-negligible and lead to quantization performance lower than floating-point computations. To obtain a non-constant quantization step, a nonlinear function must be computed, either online or offline, to generate a lookup table where operands for each operation are stored. On the other hand, uniform quantization is based on a constant quantization step. Additionally, coding only the nearest power of two does not bring an improvement on Cortex-M4-based microcontrollers. Multiplications and shifts are implemented in hardware and take only 1 cycle. In consequence, the benefits of this kind of approach are limited. For these reasons, we will rely on uniform quantization in this work.

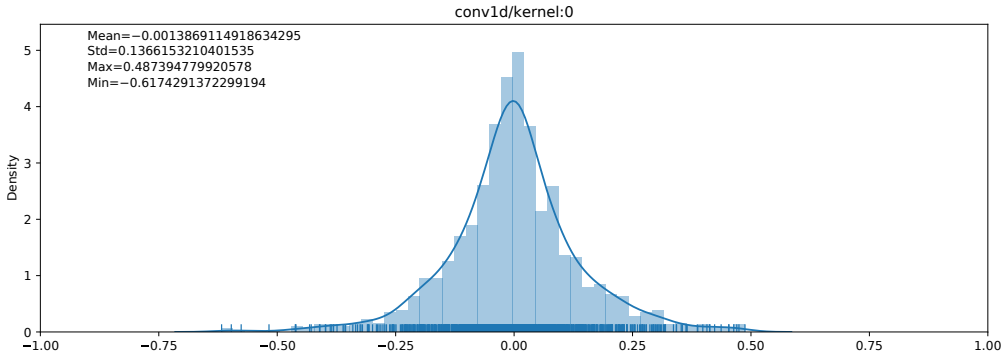

Figure 1. Example of the distribution of weights for a convolutional layer kernel.

4.1.2. Scaling and Offset

An alternative consists of finding a scale factor that is not necessarily a power of two, but scale values in $[-1; +1]$. Using this technique, all the bits (except the sign bit) are used to represent the fractional parts of numbers. As an example, there is the $Q1.15$ format for 16-bit numbers.

The scale factor also uses a fixed-point representation but with its own power of two scale factor. This allows for a slightly lower quantization error since the quantization step is not necessarily a power of two. However, scaling a number adds computations: instead of performing only a shift, a multiplication with the scale factor followed by a shift with the scale factor's power of two scale factor must be performed.

Additionally, the range could be asymmetric by introducing an offset value. It would also allow for a slightly lower quantization error when the distribution is not centered around 0. However, this requires one more addition when scaling a number. It is worth noting that using unsigned numbers for ReLU activation could help recovering one more bit for the fractional part if fully merged with the previous layer. Nevertheless, it also comes with an additional implementation complexity to perform operations between signed and unsigned numbers. All these alternatives imply a higher complexity and additional computations, with only a slight improvement of the quantization error. For these reasons, we decided to use a scale factor that is a power of two and a symmetric range.

4.1.3. Per-Network, Per-Layer and Per-Filter Scale Factor

To reach the best quantization, the scale factor should be in theory chosen for each weight value. However, storing a scale factor for each value is obviously not a reasonable approach since it leads to an important memory overhead, defeating the benefit of quantization to reduce memory usage. On the other hand, using a scale factor for the whole network is too coarse to achieve a good overall quantization error. Instead, the scale factor can be made different for each layer. Another solution consists of using a scale factor for each filter of each layer. Although more complex to implement and introducing some overhead (scale factors of the layer must be stored in memory), this approach can slightly decrease the quantization error. So far, our implementation only allows a per-network and a per-layer scale factor. The parameters and the activations can have a different scale factor.

4.1.4. Conversion Method

To convert from floating-point to fixed-point, the method starts with finding the required number of bits m to represent the unsigned integer part:

$$m = 1 + \left\lfloor \log_2 \left(\max_{1 \leq i \leq N} |x_i| \right) \right\rfloor \tag{1}$$

where x_i is an element of the floating-point vector x of length N.

A positive value of m means that m bits are required to represent the absolute value of the integer part, while a negative value of m means that the fractional part has m leading unused bits. This allows obtaining a greater precision for vectors with numbers smaller

than 2^{-1}, since the leading unused bits can be removed and replaced instead by more trailing bits for precision.

From this we can compute the number of remaining bits n for the fractional part:

$$n = w - m - 1 \tag{2}$$

where w is the data type width (e.g., 16 bits).

In this equation, 1 is subtracted to take the additional bit required to represent signed numbers into account.

A positive value of n means that n bits are available to represent the fractional part.

A negative value of n means that the fractional part cannot be represented, and the integer part cannot be represented to its full precision.

An element $xfixed_i$ of the fixed-point vector $xfixed$ is computed from the element x_i of the floating-point vector x as:

$$xfixed_i = trunc(x_i \times 2^n) \tag{3}$$

where $trunc(y)$ truncates a real number y to its integer part.

And the scale factor s is defined as:

$$s = 2^{-n} \tag{4}$$

Two methods can be used to get the weights of a deep neural network in a quantized format. These methods are detailed in the following.

4.2. Post-Training Quantization

In post-training quantization, the neural network is entirely trained using floating-point computation (a *binary32* format is assumed here). Once the training is over, the neural network is frozen, and the parameters are then quantized. The quantized neural network is then used to perform the inference, without any adjustments of the parameters.

The quantization phase introduces a quantization error on each parameter as well as on the input, thus leading to a quantization error on the activations. The accumulation of quantization errors at the output of the neural network can cause the classifier to incorrectly predict the class of the input data, creating an accuracy drop compared to the non-quantized version. As the bit width of the values decreases, the quantization error increases, and the resulting accuracy typically decreases as well. In some situations, a slight increase in the quantization error can help the network generalize better over new data, inducing a slight increase in the accuracy over test data.

4.3. Quantization-Aware Training

The objective of the Quantization-Aware Training (QAT) is to compensate the quantization error by training the deep neural network using the quantized version during the forward pass. This should help mitigating the accuracy drop to some extent. The back-propagation still relies on non-quantized values. To stabilize the learning phase with a quantized version, and then obtain better results on average, the DNN can be pre-trained using a floating-point representation in order to initialize the parameters to a sensible value.

In this work we decided to perform all the computations using a floating-point representation. As shown in Figure 2, the inputs, weights and biases of each layer are quantized (but kept in floating-point representation) before actually performing the layer's computation. The layer's output is quantized after the computation, before reaching the next layer. The quantization operation is done following the method presented in Section 4.1. During the training phase, the range of the values is reassessed each time to adjust the scale factor before performing the layer's computation. When doing inference only, the scale factor is frozen.

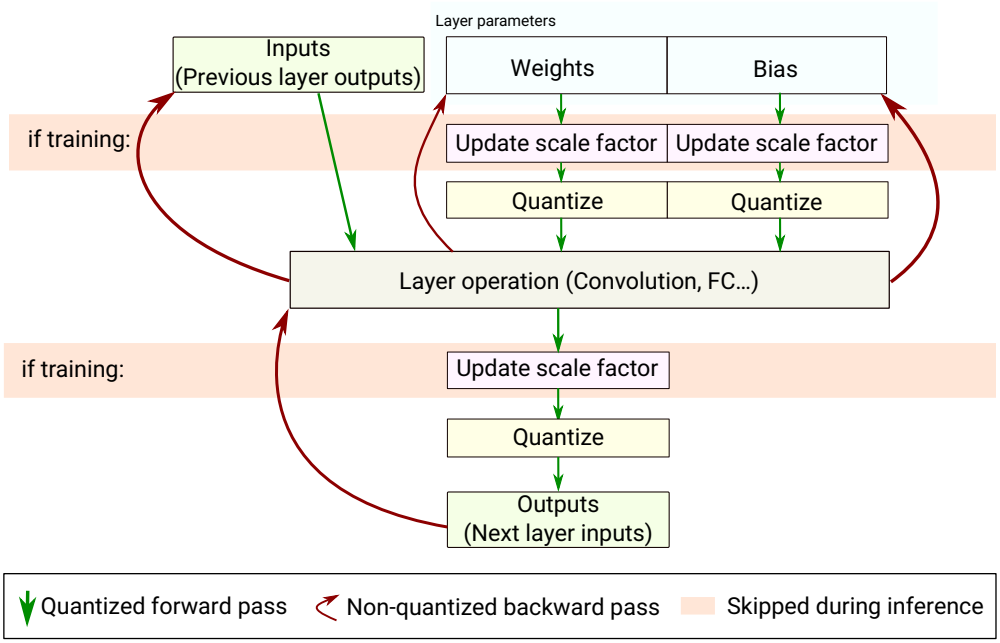

Figure 2. Quantization-Aware Training.

In case of a convolutional neural network, the convolutional and fully connected layers require a quantization-aware training for the weights. Please note that batch normalization layers also require quantization-aware training. However, as we do not use batch normalization in our experiments, it has not been implemented. Concerning max-pooling layers, nothing must be trained with quantization as they do not have weights. Moreover, as max-pooling only consists of an element-wise max, there is no need to quantize: inputs are already quantized from the previous layer and the dynamic range cannot be expanded. Actually, max-pooling can only shrink data without the possibility to recover an already lost precision. Therefore, no quantization is done on the max-pooling layers. It is similar for the ReLU activation which is considered to be a separate layer. On the other hand, the element-wise addition layer requires quantization. It does not have trainable weights; however, the dynamic range of the output can increase after adding two large numbers. Therefore, the same quantization process is applied to compute the output scale factor.

5. Deployment of Quantized Neural Network

After the network has been trained and optionally quantized, it is deployed onto a microcontroller to perform the inference on the target platform. Deployment involves the following phases:

- exporting the weights of the deep neural network and encoding them into a format suitable for on-target inference,
- generating the inference program according to the topology of the deep neural network,
- compiling the inference program,
- and then uploading the inference program with the weights onto the microcontroller's ROM.

5.1. Existing Embedded AI Frameworks

Several embedded AI frameworks are already available. Among them, the most popular ones are TensorFlow Lite for Microcontrollers [39] and STM32Cube.AI [40]. Other frameworks stemming from research projects also exist and are discussed in the following.

5.1.1. TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers (or TFLite Micro) is a project derived from TensorFlow Lite. Originally focused on deep neural network deployment on smartphones,

it has been made available for microcontrollers. TFLite Micro supports a wide range of operations [41], enabling the deployment of a variety of deep neural networks such as multi-layer perceptrons and convolutional neural networks including residual neural networks. Deep neural networks are developed and trained using TensorFlow/Keras, and can then be deployed semi-automatically onto a microcontroller.

TFLite Micro is intended to be generic enough to be deployed on any kind of 32-bit microcontroller. The inference library is therefore portable, but it also means there is no integration with specific microcontroller families and vendor tools. The trained deep neural network (topology and weights) can indeed be automatically converted to a format understandable by the inference library, but there are no tools to generate and deploy the application code. Moreover, application test must be written by hand. Nevertheless, a template source code for a few development boards (e.g., the SparkFun Edge) as well as a few demo applications (e.g., keyword spotting) are available. Finally, TFLite Micro does not come with tools to measure metrics such as the inference time or the RAM and ROM usage.

TFLite Micro supports computation on both floating-point in *binary32* format and fixed-point on 8-bit integers. The quantization technique uses a non-power of two scale factor, a symmetric range for weights and an asymmetric range for the activations. Biases are quantized on 32-bit integers. Convolution operations can make use of per-filter scale factor and offset, while other operations use per-tensor (i.e., per-layer) scale factor and offset [42,43]. There is no support for fixed-point on 16-bit integers.

Inference on 8-bit integers can be accelerated using low-level optimizations provided by the CMSIS-NN [3] library from ARM. This library uses SIMD-like instructions (from the ARMv7E-M instruction set architecture of Cortex-M4 cores) to perform two multiply–accumulate (MACC) on 16-bit operands with a single 32-bit accumulator in one cycle.

While being entirely free/open-source, the complexity of the software architecture makes it quite difficult to manipulate and extend. This is a substantial drawback in a research environment, and it also comes with additional overhead. The deep neural network topology is deployed as a sort of microcode that is interpreted at runtime instead of being statically compiled. This process makes it more difficult for the compiler to perform optimizations and causes a larger memory usage.

5.1.2. STM32Cube.AI

STM32Cube.AI is a software suite from STMicroelectronics enabling the deployment of deep neural networks onto their STM32 family of microcontrollers. STM32Cube.AI supports deployment of trained deep neural network models from several frameworks including Keras and TensorFlow Lite. A wide range of operations are supported [44], allowing the deployment of several deep neural network architectures such as multi-layer perceptron and convolutional neural networks including residual neural networks.

STM32Cube.AI is a software suite fully integrated with other STMicroelectronics development and deployment tools such as STM32CubeMX, STM32CubeIDE and STM32CubeProgrammer. This provides a very straightforward and easy to use flow. Moreover, a test application is included to evaluate the model on target with a real test dataset, providing metrics on inference time, ROM and RAM usage, without having to write a single line of code.

Like TFLite Micro, STM32Cube.AI supports computations on floating-point in *binary32* format and fixed-point on 8-bit integers. In fact, the quantization on 8-bit integers comes from TFLite. There is no support for fixed-point on 16-bit integers.

STM32Cube.AI also has an optimized inference engine that seems to be partially based on CMSIS-NN. However, as the source code of the inference engine is not freely available, it is not clear what is optimized and how.

The inference library is entirely proprietary/closed-source, therefore it is not possible to manipulate and extend this library. This represents a major drawback in a research environment. It is also not possible to use STM32Cube.AI on microcontrollers which are not

part of the STMicroelectronics portfolio. The inference process and optimizations are not detailed, but unlike TFLite Micro, the network topology is compiled into a set of function calls to the closed-source library rather than being interpreted at runtime.

5.1.3. Other Frameworks

Some other frameworks have been developed as part of research projects. These frameworks mainly focus on "classical" machine learning (SVM, Decision Tree ...) such as emlearn [45] and Micro-LM [46], or multi-layer perceptron such as Gravity [47] and FANN-on-MCU [48]. These frameworks do not support convolutional neural networks with residual connections. At the time of this work, microTVM seems less mature and popular than TensorFlow Lite for Microcontrollers and STM32Cube.AI. It is, therefore, not studied in this work.

5.2. MicroAI: Our Framework Proposition

As mentioned above, existing tools for quantized neural networks have some drawbacks that motivated the development of our own framework. This framework addresses the following issues:

- open-source frameworks do not support convolutional neural networks with non-sequential topologies,
- frameworks that support convolutional neural networks are proprietary or too complex to be modified and extended easily,
- frameworks do not provide 16-bit quantization,
- some frameworks are dedicated to a limited family of hardware targets.

In this work, we aim at providing a framework, easy to extend and modify, that allows for a complete pipeline from the neural network training to the deployment and evaluation on the microcontroller. Additionally, this framework must provide a lightweight runtime on the microcontroller to reduce the overhead. Finally, our objective is to achieve performance close to existing solutions.

Our framework, called MicroAI, is built in two parts:

1. The neural network training code that relies on Keras or PyTorch.
2. The conversion tool (called KerasCNN2C) that takes a trained Keras model and produces a portable C code for the inference

Both parts are written in Python since it is the most popular programming language to build deep neural networks and it easily interfaces with existing frameworks, libraries and tools.

5.3. MicroAI: General Flow

As seen in Figure 3, MicroAI provides an interface to automatically train, deploy and evaluate an artificial neural network model. A configuration file written in TOML [49] is used to describe the whole flow of an experiment:

- The number of iterations for the experiment (for statistical purposes)
- The dataset to use for training and evaluation
- The preprocessing steps to apply to the dataset
- The framework used for training
- The various model configurations to train, deploy and evaluate
- The configuration of the optimizer
- The post-processing steps to apply to the trained model
- The target configuration for deployment and evaluation

The three main steps, training, deployment and evaluation, are described in the following. The commands used to trigger them are available in Appendix C.

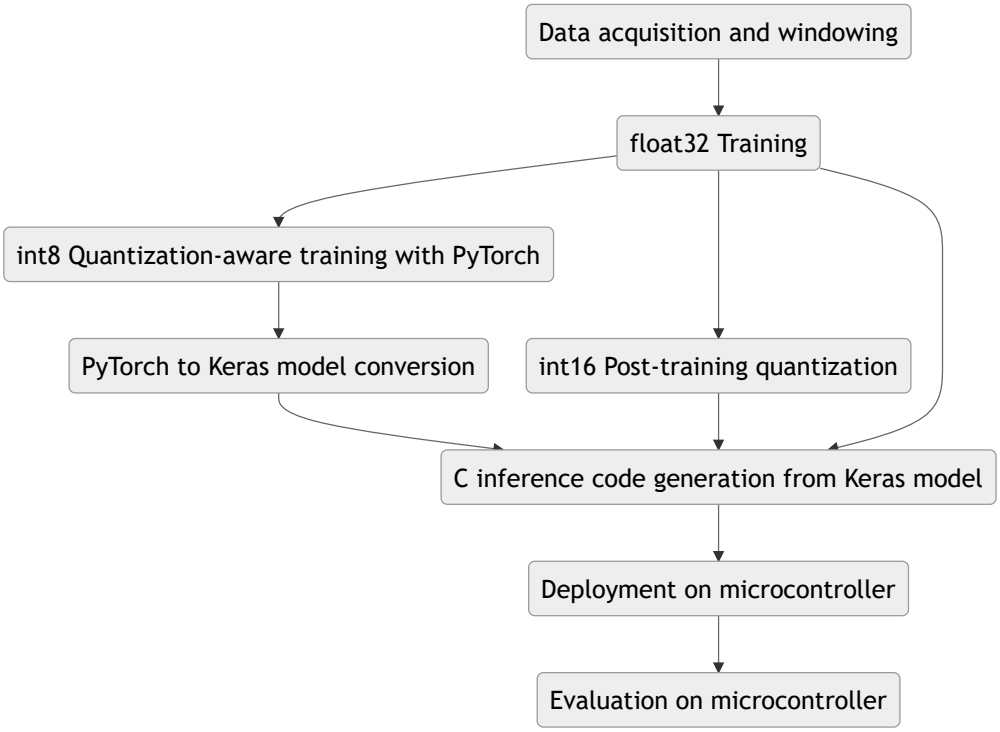

Figure 3. MicroAI general flow for neural network quantization and evaluation on embedded target.

5.4. MicroAI: Training

For the training phase, MicroAI is simply a wrapper around Keras or PyTorch.

A dataset requires an importation module to be loaded into an appropriate data model. The training process expects a `RawDataModel` instance, which gathers the train and test sets. This instance contains *numpy* arrays for the data and the labels. A higher-level data model `HARDataModel` is also available for Human Activity Recognition to process subjects and activities more easily. This model is then converted to a `RawDataModel` using the `DatamodelConverter` in the preprocessing phase. The preprocessing phase also includes features such as normalization. Dataset importation module for UCI-HAR, SMNIST and GTSRB (described in Section 6) are included and can be easily extended to new datasets.

To make use of a deep neural network architecture in the model configuration, it must be first described according to the training framework API in use. The description of the model is a template where parameters can be set by the user in the configuration file. MicroAI provides the following neural network architectures for both Keras and PyTorch:

- MLP: a simple multi-layer perceptron with configurable number of layers and neurons per layer.
- CNN: a 1D or 2D convolutional neural network with configurable number of layers, filters per layer, kernels and pools size, and number of neurons per fully connected layer for the classifier
- ResNet: a 1D or 2D residual neural network (v1) with convolutional layers. The number of blocks and filters per layer, stride, kernel size, and optional BatchNorm can be configured.

All architectures use ReLU activation units.

In the configuration file, several model settings can be described, each inside their own `[[model]]` block. Each model will be trained sequentially. A common configuration for all the models can be specified in a `[model_template]` block. Model configuration also includes optimizer configuration and other parameters such as the batch size and the number of epochs.

Once the model is trained, some post-processing can be applied. It is for instance possible to remove the SoftMax layer for Keras models with the `RemoveKerasSoftmax` module. This layer is indeed useless when only inference is performed.

Even though it also performs model training, Quantization-aware training described in Section 4.3 is also included for PyTorch as a post-processing step in the `QuantizationAwareTraining` module. The actual training step before post-processing is seen as a general training, before optionally performing post-training quantization or quantization-aware training. The quantization-aware training can be seen as a fine-tuning on top of the more general training (which can also be skipped if necessary). The quantization-aware training does not actually convert the weights from a floating-point data type to an integer data type with fixed-point representation. This conversion is rather performed by the KerasCNN2C conversion tool or another deployment tool.

Support for additional learning frameworks can be added by creating a new class implementing the `LearningFramework` interface and by supplying compatible model templates.

5.5. MicroAI: Deployment

MicroAI can deploy a trained model to perform inference on a target using either STM32Cube.AI, TensorFlow Lite for Microcontrollers or our own tool, KerasCNN2C.

STM32Cube.AI can be used for STM32 platforms, support for the Nucleo-L452RE-P with an STM32L452RE microcontroller is included. Support for other platforms using a STM32 microcontroller can be added by providing a sample STM32CubeIDE project including the X-CUBE-AI package. STM32Cube.AI does not support microcontrollers outside the STM32 family.

TensorFlow Lite for Microcontrollers is a portable library that can be included in any project. Therefore, it could be used for any 32-bit microcontroller. However only integration with the SparkFun Edge platform with an Ambiq Apollo3 microcontroller is included in our framework so far.

Similarly, KerasCNN2C produces a portable library that can be included in any project. So far, only integration with the Nucleo-L452RE-P and the SparkFun Edge boards has been performed. Support for other platforms can be added by providing project files calling the inference code and a module interfacing with the build and deployment tools for that platform.

Please note that none of these tools can take a PyTorch trained model as an input to deploy onto a microcontroller. The PyTorch trained model must therefore be converted to a Keras model prior to the deployment. Our framework provides a module to perform semi-automatic conversion from a PyTorch model to a Keras model. A Keras model that matches the structure of the PyTorch model must be programmed and the matching between the PyTorch model and Keras model layer names also must be specified. The semi-automatic conversion module can then automatically copy the weights from the PyTorch model to the Keras model and export it for use by one of the deployment tool.

5.6. KerasCNN2C: Conversion Tool from Trained Keras Model to Portable C Code

KerasCNN2C is a tool that we developed to automatically generate, from a trained Keras model exported as an `HDF5` file, a C library for the inference. It can also be used independently of the MicroAI framework.

In this work, only 1D models are evaluated on target. Work is underway for full support of 2D models deployment. Training and quantization are already supported, therefore 2D models are evaluated offline. Here are the supported layers so far:

- Add
- AveragePooling1D
- BatchNormalization
- Conv1D
- Dense
- Flatten

- MaxPooling1D
- ReLU
- SoftMax
- ZeroPadding1D

Layers can have multiple inputs such as the *Add* layer, thus allowing residual neural networks (ResNet) to be built. The sequential convolutional neural network or multi-layer perceptron models are also supported.

The generated library exposes a function in the `model.h` header to run the inference process with the following signature:

```
void cnn(
const number_t input[MODEL_INPUT_CHANNELS][MODEL_INPUT_SAMPLES],
output_layer_type output);
```

where `number_t` is the data type used during inference defined in the `number.h` header, `MODEL_INPUT_CHANNELS` and `MODEL_INPUT_SAMPLES` are the dimensions of the input defined in the generated `model.h` header. The input and output arrays must be allocated by the caller.

The model inference function does not proceed to the conversion of the input from fixed-point to floating-point representation when using a fixed-point inference code. The caller must perform the conversion before feeding the buffer to the model inference function (see Section 5.8).

To this aim, a floating-point number `x_float` can be converted to a fixed-point number `x_fixed` with the following call:

```
x_fixed = clamp_to_number_t((long_number_t)(x_float*(1<<INPUT_SCALE_FACTOR)));
```

where `long_number_t` is a type twice the size of `number_t` and `clamp_to_number_t` saturates and converts to `number_t`. Both are defined in the `number.h` header. `INPUT_SCALE_FACTOR` is the scale factor for the first layer, defined in the `model.h` header.

The output array corresponds to the output of the model's last layer, which is typically a fully connected layer when solving a classification problem. If the purpose is to predict a single class, the caller must find the index of the max element in the output array.

5.7. KerasCNN2C: Conversion Process

This tool first parses the model using the Keras API from TensorFlow 2.4 and generates an internal representation of the topology (i.e., a graph), with each node corresponding to a layer.

Then, a series of transformations is performed to produce a graph better suited for deployment on a microcontroller:

- Combine ZeroPadding1D layers (if they exist) with the next Conv1D layer
- Combine ReLU activation layers with the previous Conv1D, MaxPooling1D, Dense or Add layer
- Convert BatchNorm [50] weights from the mean μ, the variance V, the scale γ, the offset β and ϵ to a multiplicand w and an addend b using the following formula:

$$w = \frac{\gamma}{\sigma} \tag{5}$$

$$\sigma = \sqrt{V + \epsilon} \tag{6}$$

$$b = \beta - \frac{\gamma \times \mu}{\sigma} \tag{7}$$

so that the output of the BatchNorm layer can be computed as $y = w \times x + b$. It could be folded in the previous convolutional layer, but this is not implemented yet.

Then, for each node in the graph, the weights of the layer go through the quantization and conversion module if the conversion to fixed-point representation is enabled. The C

inference function is generated from a Jinja2 [51] template file using the layer's configuration. Similarly, the layer's weights are converted into a C array from a Jinja2 template file. Code generation is used to avoid runtime overhead of an interpreter such as the one used in TensorFlow Lite for Microcontrollers. Additionally, it allows the compiler to perform better optimizations. Indeed, the layer's configuration is generated as constants or literals in the code, allowing the compiler to perform appropriate optimizations such as loop unrolling, using immediates when relevant and doing better register allocation. By default, GCC's `-Ofast` optimization level is enabled. Moreover, the code is written in a simple and straightforward way. So far, no special effort has been made to further optimize the source code for faster execution.

The allocator module aims at saving the RAM usage. To do so, it allocates the layer's output buffers in the smallest number of pools without conflicts. For each layer of the model, its output buffer is allocated to the first pool that satisfies two conditions: it must neither overwrite its input, nor the output of a layer that has not already been consumed. If there is no such available pool, a new one is created. It is worth noting that the allocator module does not yet try to optimize the allocation to minimize the size of each pool (this is a harder problem to solve). In consequence, the total RAM usage is not optimized.

Finally, the main function `cnn(...)` is generated. This function only contains the allocation of the buffers done by the allocator module and a sequence of calls to each of the layers' inference function. The correct input and output buffers are passed to each layer according to the graph of the model.

5.8. KerasCNN2C: Quantization and Fixed-Point Computation

The post-training quantization is performed by the quantization module itself. The scale factor for each layer is found according to the method in Section 4.1.4, but it can also be specified manually for the whole network. The fixed-point coding used for all the weights is computed according to this method as well, and the data type is converted from `float` to an integer data type, such as `int8_t` for 8-bit quantization or `int16_t` for 16 bits quantization.

When doing quantization-aware training, the scale factors are found during the training phase (also according to the method in Section 4.1.4 as previously explained). Therefore, the quantization module reuses them. However, the weights are still in floating-point representation since the training phase only relies on floating-point computation. In consequence, the quantization module must perform a data type conversion similar to the one performed for post-training quantization.

Once the model is deployed and running on the target, the fixed-point computation can be done using a regular integer arithmetic and logic unit. The data type for the input and output of a layer is the same as the one used to store the weights. To avoid overflows, computation is done using a data type twice the width of the operands' data type. For example, if the data type of the weights and inputs is `int16_t`, then the intermediate results in a layer are computed and stored with an `int32_t` data type. The result is then scaled back to the correct output scale factor before saturating and converting it back to the original operands' data type.

Before performing an addition or a subtraction, operands must be represented with the same number of integer and fractional bits. It is not required for the multiplication, but the result's number of bits allocated for the fractional part is the sum of the number of bits for the fractional part of both operands. Therefore, after a multiplication, the result must be scaled to the required format by shifting the result to the right by the appropriate number of bits.

In Appendix E, the number of operations required for the main layers of a residual neural network in our implementation are provided, along with the number of cycles taken for these operations. Enabling compiler optimizations generates some ARMv7E-M instructions, namely `SMLABB` that performs a multiply–accumulate operation in one cycle (instead of two cycles). However, the compiler does not make use of the `SSAT` operation

that could allow saturating in one cycle. Instead, it uses the same instructions as a regular max operation, i.e., a compare instruction and a conditional move instruction requiring a total of two cycles.

6. Results

All the results presented in this section rely on the same model architecture, a ResNetv1-6 network with the layers shown in Figure 4. The number of filters per layer f is the same for all layers, but is modified to adjust the number of parameters of the model. The convolutional and pooling layers are one-dimensional except when handling the GTSRB dataset for which they are two-dimensional.

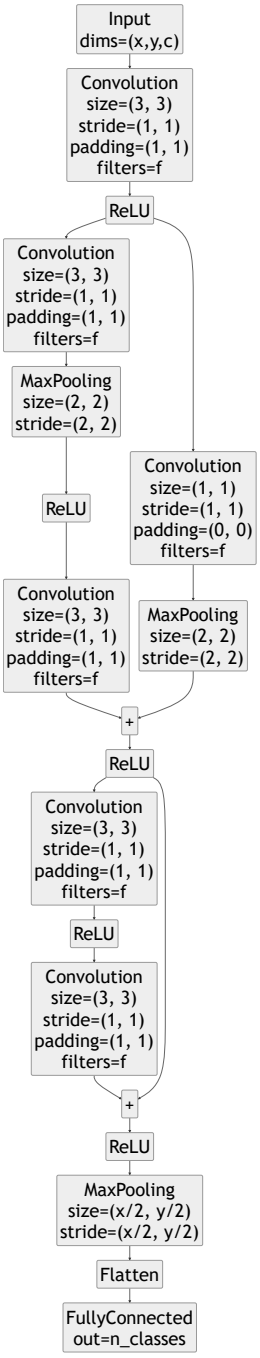

Figure 4. ResNet model architecture.

For each experiment, the residual neural network is initially trained using 32-bit floating-point numbers (i.e., without quantization), and then evaluated over the testing set. This baseline version is depicted as *float32* in the figures shown in the following.

The *float32* neural network is quantized for fixed-point on 16-bit integers inference and is then evaluated without additional training. This version is depicted as *int16* in the figures shown hereafter. Quantization is performed using the Q7.9 format for the whole network, meaning the number of bits n for the fractional part is fixed to 9.

The *float32* neural network is also trained and evaluated for fixed-point on 8-bit integers inference using quantization-aware training. This version is indicated as *int8* in the figures. In this case, the fixed-point precision can vary from layer to layer and is determined using the method introduced in Section 4.1.4.

The SGD optimizer is used for all experiments. The stability of the SGD optimizer has motivated this choice, especially for the quantization-aware training. Training parameters are described below for each dataset. Additionally, training and testing sets are normalized using the z-score of the training set. It is worth noting that Mixup [52] is also used during training.

Accuracy is not evaluated directly on the target due to the amount of time it would require. Only inference time for the UCI-HAR dataset is measured on the target.

The figures show an average over 15 runs for each point.

6.1. Evaluation of the MicroAI Quantization Method

6.1.1. Human Activity Recognition dataset (UCI-HAR)

The University of California Irvine's hosted Human Activity Recognition dataset (UCI-HAR) [53] is a dataset of activities of daily living recorded using the accelerometer and gyroscope sensors of a smartphone. In this experiment, we use the raw data coming from the sensors and divided in fixed time windows rather than the pre-computed features. The reason is that we want to perform real-time embedded recognition. To do so, it is required to avoid the overhead of computing the features for each inference before entering the deep neural network. Instead, the features are extracted by the convolutional neural network itself.

The dataset is divided into a training set of 7352 vectors and a testing set of 2947 vectors. Each vector is a one-dimensional time series of 2.56 s composed of 128 samples sampled at 50 Hz, with 50% overlap between vectors. Each sample has 9 channels: 3 axes of total acceleration, 3 axes of angular velocity and 3 axes of body acceleration. Six different classes are available in the dataset: walking, walking upstairs, walking downstairs, sitting, standing and lying.

The initial training without quantization is performed using a batch size of 64 over 300 epochs. Initial learning rate is set to 0.05, momentum is set to 0.9 and weight decay is set to 5×10^{-4}. The learning rate is multiplied by 0.13 at epochs 100, 200 and 250. The quantization-aware training for fixed-point on 8-bit integers uses the same parameters.

As can be seen in Figure 5, for the UCI-HAR dataset, the same accuracy is obtained using a 16-bit quantization (*UCI-HAR int16*) or 32-bit floating-point (i.e., the baseline *UCI-HAR float32*), whatever the number of filters per convolution.

On the other hand, we can observe that the 8-bit quantization causes an accuracy drop that increases up to 0.81% when the number of filters per convolution grows, even though quantization-aware training is used to mitigate this issue.

In Figure 6, we can observe that the accuracy obtained using 8-bit and 16-bit quantization is similar only for deep neural networks exhibiting a reduced number of parameters, in other words a low memory footprint.As an example, for 16 filters per convolution, an 8-bit quantization leads to an accuracy of 92.41% while requiring 3958 memory bytes to store the parameters. When a 16-bit quantization is used, an accuracy of 92.46% can be achieved, but at the cost of an increase of the required memory for storing the parameters (7916 bytes).

As can be seen, when more than 24 filters per convolution are used, the 16-bit quantization clearly exhibits the best accuracy vs. memory ratio. For more than 48 filters per convolution, the 8-bit quantization even provides a worse ratio than the baseline.

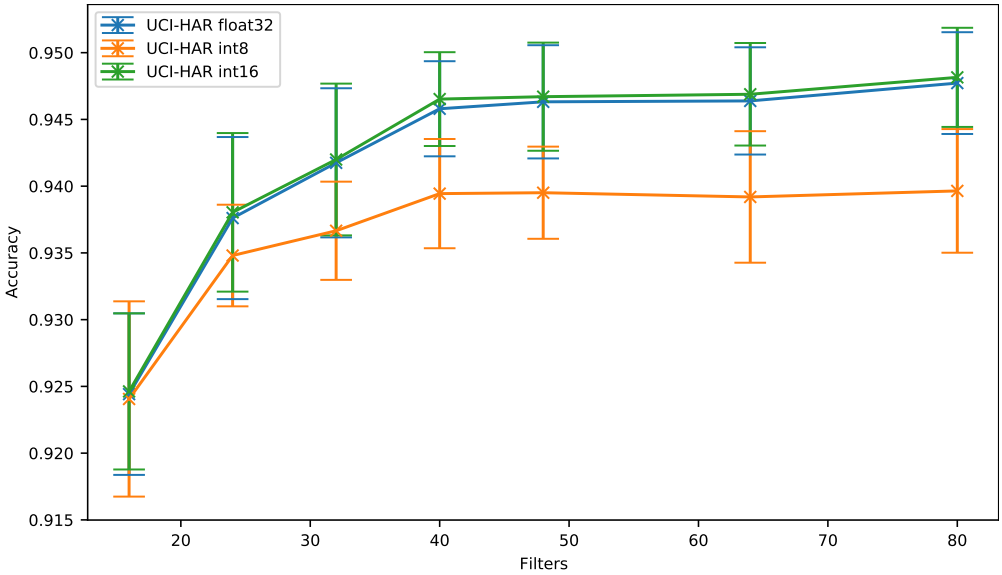

Figure 5. Human Activity Recognition dataset (UCI-HAR): accuracy vs. filters.

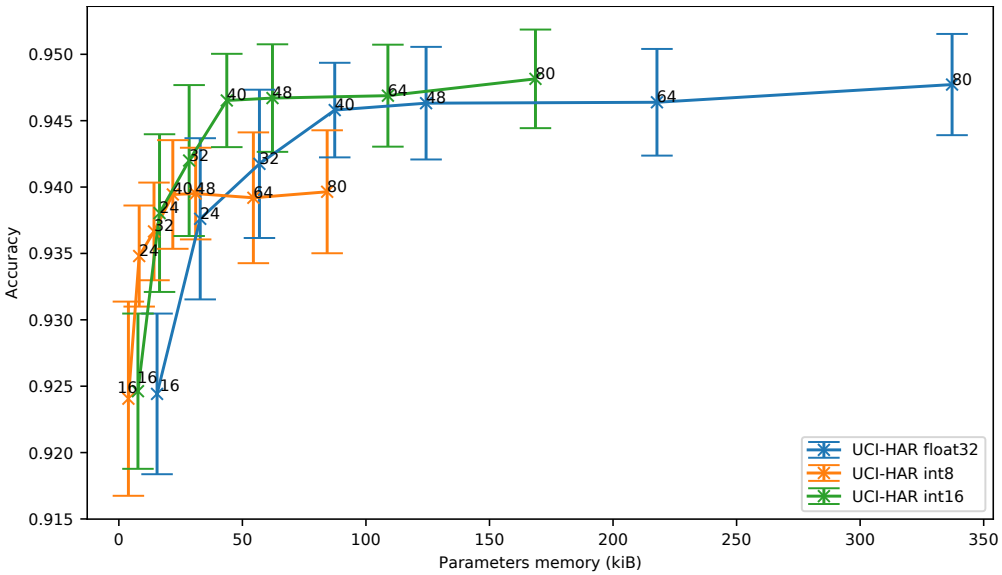

Figure 6. Human Activity Recognition dataset (UCI-HAR): accuracy vs. parameters memory.

6.1.2. Spoken Digits Dataset (SMNIST)

Spoken MNIST is the spoken digits part of the written and spoken digits database for multi-modal learning [54]. This dataset is made of spoken digits extracted from the Google Speech Commands [55] dataset. The audio signal is preprocessed to obtain 12 MFCC plus an energy coefficient using a window of 50 ms with 50% overlap over the audio files of approximately 1 s each, generating one-dimensional series of 39 samples with 13 channels. The dataset is divided into a training and a testing set of 34,801 and 4107 vectors, respectively. Some samples are duplicated to obtain 60,000 training vectors and 10,000 testing vectors. There are 10 different classes for each digit between 0 and 9.

The initial training, without quantization, uses a batch size of 256 over 120 epochs. Initial learning rate is set to 0.05, momentum is set to 0.9 and weight decay is set to 5×10^{-4}. Learning rate is multiplied by 0.1 at epochs 40, 80 and 100.

The quantization-aware training for fixed-point on 8-bit integers uses a batch size of 1024 over 140 epochs. Initial learning rate, momentum and weight decay are the same as for the initial training. Learning rate is multiplied by 0.1 at epochs 40, 80, 100 and 120.

As can be observed in Figure 7 and regardless of the number of filters, the 16-bit quantization (*SMNIST int16*) provides overall a similar accuracy compared to the floating-point baseline (*SMNIST float32*). On the other hand, the accuracy drops by up to 1.07% when the 8-bit quantization is used. However, the accuracy drop slightly decreases when 48 filters per convolution are used, and then stays around 0.5% and 0.6% for a higher number of filters.

In Figure 8, we can see that the 16-bit quantization is still the best solution in terms of memory footprint. Despite the fact that the 8-bit quantization stays closer to 16-bit quantization on SMNIST than on UCI-HAR, the 8-bit quantization does not provide any benefit over 16-bit quantization in terms of accuracy vs. memory ratio even for small neural networks.

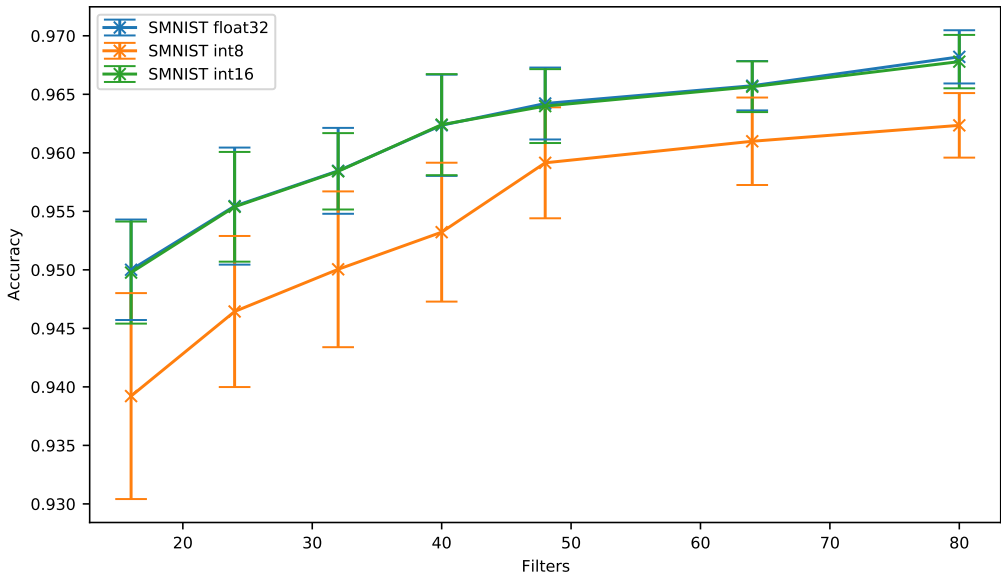

Figure 7. Spoken digits dataset (SMNIST): accuracy vs. filters.

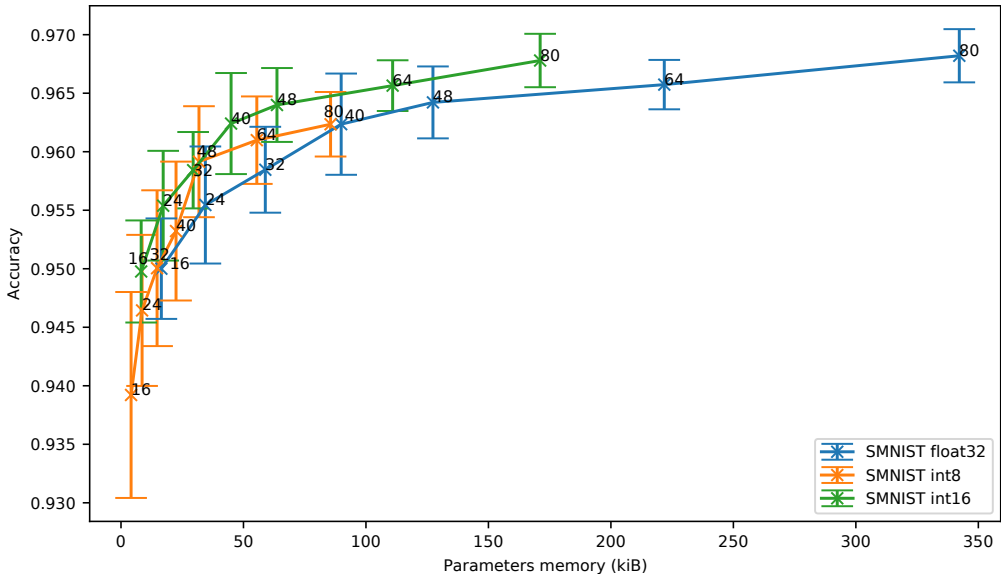

Figure 8. Spoken digits dataset (SMNIST): accuracy vs. parameters memory.

6.1.3. The German Traffic Sign Recognition Benchmark (GTSRB)

The German Traffic Sign Recognition Benchmark (GTSRB [56]) is a dataset containing various road signs color pictures. Image sizes vary between 15×15 to 250×250 pixels. In this experiment, the two-dimensional images are scaled to 32×32 pixels using bilinear interpolation and anti-aliasing, while keeping the 3 color channels (red, green, blue). The dataset is divided into a training and a testing set of 39,209 and 12,630 vectors, respectively. There are 43 different classes, one for each type of road sign in the dataset.

The initial training without quantization uses a batch size of 128 over 120 epochs. Initial learning rate is set to 0.01, momentum is set to 0.9 and weight decay is set to 5×10^{-4}. Learning rate is multiplied by 0.1 at epochs 40, 80 and 100.

The quantization-aware training for fixed-point on 8-bit integers uses a batch size of 512 over 120 epochs. Initial learning rate, momentum and weight decay are the same as for the initial training. Learning rate is multiplied by 0.1 at epochs 20, 60, 80 and 100.

The accuracy results obtained for 8- and 16-bit quantization and the 32-bit floating-point versions are shown in Figure 9 for different number of filters. As can be seen, the 16-bit quantization (*GTSRB int16*) provides an accuracy similar to the one obtained with the baseline (*GTSRB float32*). In the meantime, an accuracy drop of up to 1.1% can be observed when the 8-bit quantization is used with this GTSRB dataset. However, and as it has been observed with the SMNIST dataset, the accuracy gets closer to the baseline when the network has more filters (0.33% for 64 filters).

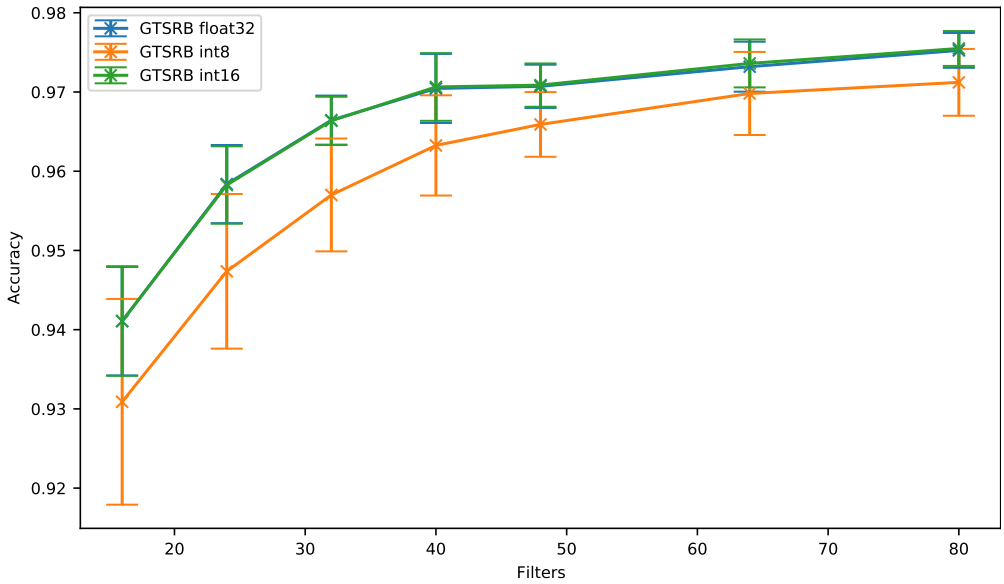

Figure 9. German Traffic Sign Recognition Benchmark: accuracy vs. filters.

Moreover, even though the 8-bit quantization does not outperform the results obtained with the 16-bit quantization, the Figure 10 shows that the 8-bit quantization can represent an interesting solution when a two-dimensional network is used on an image dataset.

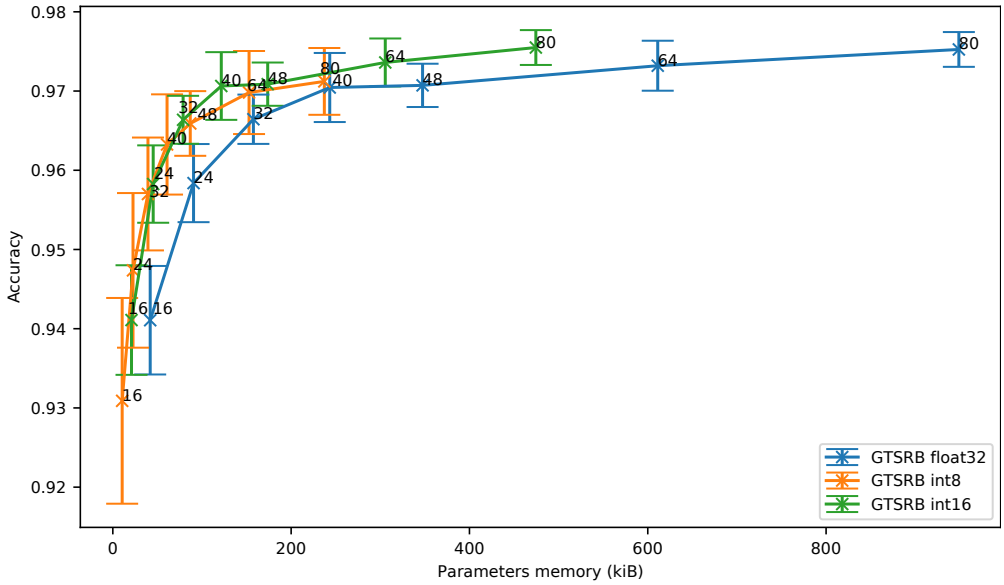

Figure 10. German Traffic Sign Recognition Benchmark: accuracy vs. parameters memory.

6.2. Evaluation of Frameworks and Embedded Platforms

In our experiments, two different targets have been used to deploy a deep neural network on a microcontroller: the SparkFun Edge and the Nucleo-L452RE-P. Both platforms are set to run at 48 MHz on a 3.3 V supply and their main specifications are summarized in Table 3.

Table 3. Embedded platforms.

Board	Nucleo-L452RE-P	SparkFun Edge
MCU	STM32L452RE	Ambiq Apollo3
Core	Cortex-M4F	Cortex-M4F
Max Clock	80 MHz	48 MHz (96 MHz "Burst Mode")
RAM	128 kiB	384 kiB
Flash	512 kiB	1024 kiB
CoreMark/MHz	3.42	2.479
Run current @3.3 V, 48 MHz	4.80 mA	0.82 mA *

* After removing peripherals (Mic1&2, accelerometer ...).

V_{DD_MCU} is set to 1.8 V for the Nucleo-L452RE-P platform and current measurement is taken from the I_{DD} jumper. It does not have any on-board peripherals. On the SparkFun Edge board, the measure of the current is done using the power input pin of the board (after the programmer). The built-in peripherals were unsoldered from the board to eliminate their power consumption. The current consumption was measured using a Brymen BM857s auto-ranging digital multimeter configured in max mode. The energy results are based on this maximum observed current consumption and the supply voltage of 3.3 V.

As can be seen in Table 3, and even though both platforms are built around a Cortex-M4F core running at the same frequency, thanks to its subthreshold operation the SparkFun Edge board consumes much less power than the Nucleo-L452RE-P, while having also more Flash and RAM memory. However, results obtained with the CoreMark benchmark show that the Ambiq Apollo3 microcontroller is slower than the STM32L452RE. It is worth noting that the CoreMark results have been measured on the Ambiq Apollo3 microcontroller, while it has been extracted from the datasheet for the STM32L452RE microcontroller.

The deep neural network used in our experiments is the residual neural network described in Section 6. This network has been trained on the UCI-HAR dataset presented in Section 6.1.1. Inference time is measured from 50 test vectors from the testing set of UCI-HAR on the microcontrollers. TensorFlow Lite for Microcontrollers version 2.4.1 has been used to deploy the deep neural network on the SparkFun Edge board, while STM32Cube.AI version 5.2.0 has been used to deploy it on the Nucleo-L452RE-P board, both for the 32-bit floating-point and fixed-point on 8-bit integers inference. Our framework is used to deploy the deep neural network on both platforms for 32-bit floating-point, fixed-point on 16-bit integers and fixed-point on 8-bit integer inference. It is worth noting that optimizations for the Cortex-M4F provided by CMSIS-NN are enabled for both TensorFlow Lite for Microcontrollers and STM32Cube.AI tools. Our framework does not make use of these optimizations yet. The main characteristics of the frameworks are summarized in Table 4.

Table 4. Embedded AI frameworks.

Framework	**STM32Cube.AI**	**TFLite Micro**	**MicroAI**
Source	Keras, TFLite, ...	Keras, TFLite	Keras, PyTorch *
Validation	Integrated tools	None	Integrated tools
Metrics	RAM/ROM footprint, inference time, MACC	None	ROM footprint inference time
Portability	STM32 only	Any 32-bit MCU	Any 32-bit MCU
Built-in platform support	STM32 boards (Nucleo, ...)	32F746GDiscovery, SparkFun Edge, ...	SparkFun Edge, Nucleo-L452-RE-P
Sources	Private	Public	Public
Data type	`float`, `int8_t`	`float`, `int8_t`	`float`, `int8_t`, `int16_t`
Quantized data	Weights, activations	Weights, activations	Weights, activations
Quantizer	Uniform (from TFlite)	Uniform	Uniform
Quantized coding	Offset and scale	Offset and scale	Fixed-point *Qm.n*

* PyTorch models must be semi-automatically converted to Keras model prior to deployment.

To compare software and hardware platforms, only the results with 80 filters per convolution are analyzed below. Nevertheless, results with less than 80 filters are still available in the tables of Appendix D to highlight how fast and efficient a small deep neural network can be when deployed on a constrained embedded target. They also highlight a higher overhead for very small neural networks especially for TensorFlow Lite for Microcontrollers compared to our framework.

In Figure 11, we can observe that TFLite Micro has a higher overhead than STM32Cube.AI, while MicroAI exhibits a slightly lower overhead than STM32Cube.AI. As outlined in Table A3 of Appendix D, when the number of filters per convolution increases, most of the ROM is used by the model's weights.

The inference time obtained for both platforms and the different deployment tools is illustrated in Figure 12. As can be seen, the STM32Cube.AI with the 8-bit inference provides the best solution as it requires only 352ms for one inference. In the same configuration, TensorFlow Lite for Microcontrollers requires 592ms for one inference. Finally, 1034 ms and 1003 ms are required for one inference using our framework on the Nucleo-L452RE-P board and the SparkFun Edge board, respectively.

When using fixed-point on 16-bit integers for the inference, our framework provides approximately the same performance than with 8 bits. The reason is that the inference code is the same: similar instructions are generated, and computations are still performed using 32-bit registers. On the Nucleo-L452RE-P, we can observe that the inference time for one input is 1223 ms, while it is only 1042 ms on the SparkFun Edge board. We guess

this improvement should be due to a different implementation around the core in terms of memory access, especially the cache for the Flash memory.

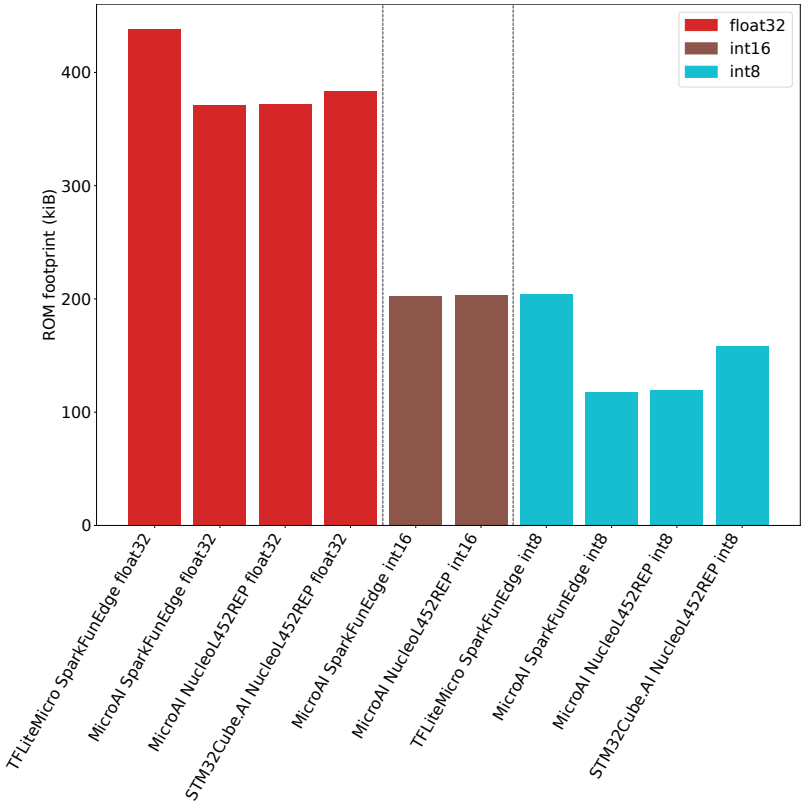

Figure 11. ROM footprint for TFLite Micro, STM32Cube.AI and MicroAI with 80 filters per convolution.

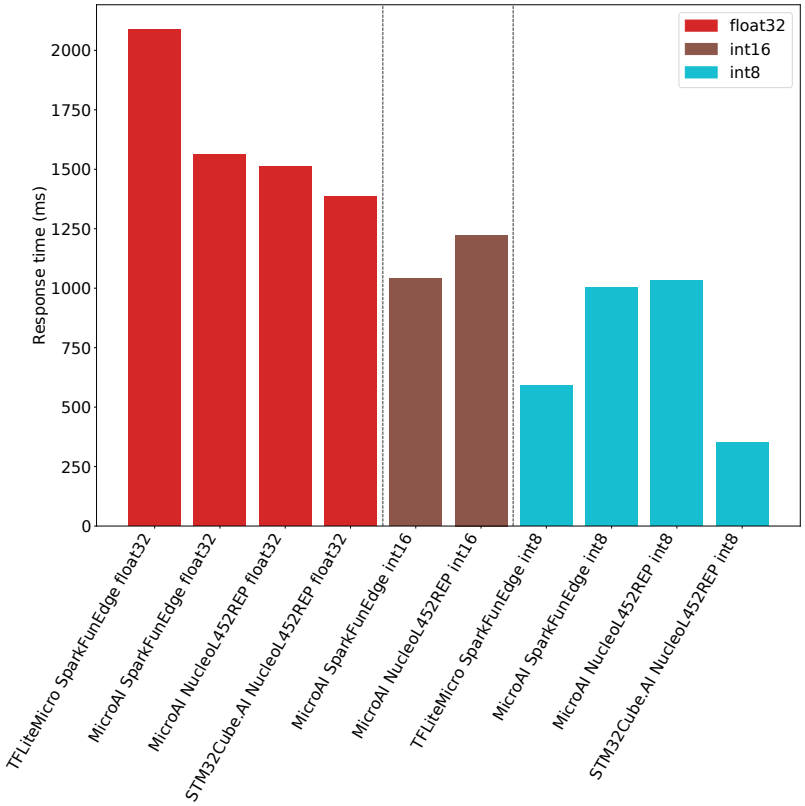

Figure 12. Inference time for 1 input for TFLite Micro, STM32Cube.AI and MicroAI with 80 filters per convolution.

The Figure 12 also shows that whatever the tool and target, the 32-bit floating-point inference is slower than 16- or 8-bit quantization. We can also observe that our framework requires 1561 ms and 1512 ms for one inference on the SparkFun Edge and the Nucleo-L452RE-P boards, respectively. In the meantime, the STM32Cube.AI requires 1387 ms for one inference on the Nucleo-L452RE-P board. Our framework exhibits therefore performance comparable to the STM32Cube.AI. Finally, we can see that TensorFlow Lite for microcontrollers on the SparkFun Edge board provides lower performance as it requires 2087 ms to perform one inference

To conclude, and as outlined in Figure 13, we can say the SparkFun Edge board provides the best power efficiency in all situations. The reason is that the SparkFun Edge board power consumption is approximately 6 times lower compared to the Nucleo-L452RE-P.Using the SparkFun Edge board and TensorFlow Lite for Microcontroller with fixed-point on 8-bit integers, one inference requires 0.45 µWh of energy consumption. On the other hand, our framework requires 0.75 µWh and 0.78 µWh on the SparkFunEdge board for fixed-point on 8-bit and 16-bit integers inference, respectively. When 32-bit floating-point is used for inference on the SparkFun Edge board, our framework provides a better energy efficiency than TensorFlow Lite for Microcontrollers as it requires 1.17 µWh instead of 1.57 µWh.

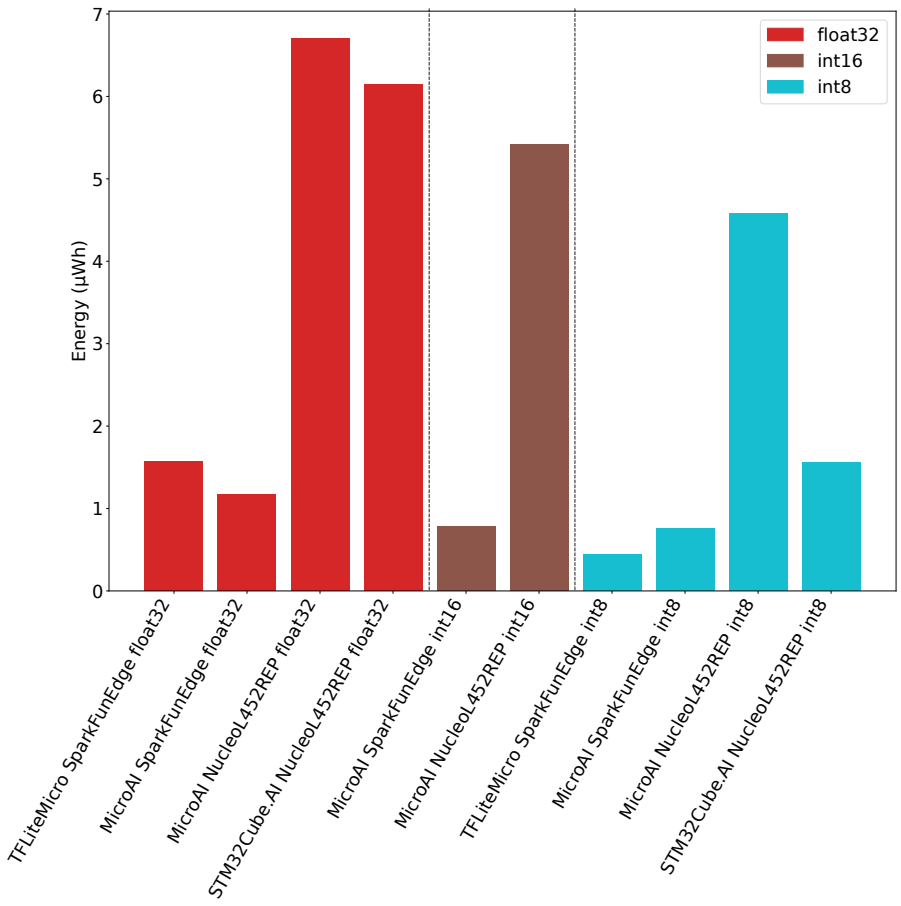

Figure 13. Energy consumption for 1 input for TFLite Micro, STM32Cube.AI and MicroAI with 80 filters per convolution.

Concerning the energy consumed on the Nucleo-L452RE-P board, our framework requires 4.58 µWh, 5.42 µWh and 6.70 µWh for one inference using fixed-point on 8-bit integers, 16-bit integers and 32-bit floating-point, respectively. In the meantime, only 6.15 µWh of energy are required for one inference when the STM32Cube.AI framework is used with 32-bit floating-point. Finally, we can see that the required energy for one inference when using STM32Cube.AI with fixed-point on 8-bit integers is 1.56 µWh on the Nucleo-L452RE-P.

This amount of energy is similar to the one obtained with TensorFlow Lite for Microcontrollers on the SparkFun Edge board when performing floating-point inference.

7. Discussion

First, a high variance can be observed when we compare the accuracy results obtained on the three datasets versus the model size. This variability makes it difficult to draw any definitive conclusions. However, there is a trend in our results that allows getting some insights of the performance for each experiment.

As has been shown, execution using fixed-point on 8-bit and 16-bit integers provide a significant decrease of the inference time, thus reducing as well the average power consumption. As power consumption is a key parameter in embedded systems, having a lower inference time is interesting as it is possible either to reduce the microcontroller's operating frequency or to put the microcontroller in sleep mode for a longer period between two inferences. Additionally, execution using 8-bit and 16-bit integers also provide a significant reduction in memory footprint. The required memory for the model parameters is divided by 4 and 2 for for 8-bit and 16-bit quantization, respectively. It is worth noticing that the RAM usage, which is not illustrated here, is also reduced.

Our results also show that performing inference using quantization for fixed-point on 16-bit integers does not lead to an accuracy drop whatever the considered test case. Moreover, inference using 16-bit does not require quantization-aware training to achieve such results. As both the power consumption and the memory footprint can be decreased, fixed-point quantization on 16-bit integers can therefore always be preferred to a 32-bit floating-point inference.

On the other hand, 8-bit quantization does not provide a substantial improvement over 16-bit quantization. Moreover, 8-bit quantization requires performing a quantization-aware training to even achieve the obtained results. It is worth noting that quantization-aware training for 8-bit quantization introduces more variance in the results over the baseline, and is also more sensitive to a change in the training parameters. As it is quite difficult to achieve a stable training, it is preferable to use an optimizer such as SGD with conservative parameters, instead of optimizers such as Adam or RAdam, to reduce the variance of the results, even though it means achieving a lower maximum accuracy.

During our experiments, it has also been observed that the 8-bit Post-Training Quantization of TensorFlow Lite achieves better results compared to the 8-bit Quantization-Aware Training provided by our framework. This is likely due to the combination of per-filter quantization, asymmetric range, non-power of two scale factor as well as optimizations of TensorFlow Lite to avoid unnecessary truncation, therefore loss of precision. We also observed that using 9-bit instead of 8-bit during the Post-Training Quantization allows us to outperform the TensorFlow Lite quantization performance. Some results showing this improvement are available in Appendix B for the UCI-HAR dataset. From these results, we can conclude that the slight additional precision brought by the combination of per-filter quantization, asymmetric range and non-power of two scale factor does in fact matter. Implementing these methods in our framework seems therefore required to reduce the accuracy loss of our 8-bit quantization.

Another benefit of 8-bit quantization is that SIMD instructions can be used (with some classes of microcontrollers) to improve the inference time and then further reduce the power consumption. Such instructions allow performing in a single cycle either 2 multiply–accumulate with common accumulator of 16-bit operand (*SMLAD*), or 2 additions of 16-bit operand (*QADD16*), or shift and saturation in (*SSAT*). In the case of inference using 16-bit integers, it is not always possible to make use of such instructions since intermediate results must be stored in 32-bit and some of these instructions require 16-bit operands. The *SMLAD*, *QADD16* and *SSAT* instructions are not yet used in our framework, but it is a work on progress. Nonetheless, a 16-bit quantization scheme can be used with our framework, which is not the case with either TensorFlow Lite for Microcontrollers or STM32Cube.AI. The 16-bit quantization from our framework provides a good compromise

between accuracy, inference time and memory footprint without requiring additional work on quantization-aware training as presented in the previous results.

The results obtained on inference time clearly show that both the software and the hardware platforms have a substantial impact on energy efficiency. STM32Cube.AI offers the most optimized inference engine in terms of execution time, both in floating-point and fixed-point on integers. Our results show that TensorFlow Lite for Microcontrollers is slower than STM32Cube.AI in both conditions. For the floating-point inference, our framework is in between these two software platforms as it is only slightly slower than STM32Cube.AI. However, as optimizations using SIMD instructions have not been implemented yet in our framework, inference using 8-bit integers still provides lower performance than TensorFlow Lite for Microcontrollers and STM32Cube.AI.

Regardless of the software performance, running STM32Cube.AI on a Nucleo-L452RE-P board is only competitive with inference using 8-bit integers when compared to TensorFlow Lite for Microcontrollers in 32-bit floating-point inference on the SparkFun Edge board. The reason is that the Ambiq Apollo3 microcontroller on the SparkFun Edge board is much more energy efficient. In all remaining cases, running TensorFlow Lite for Microcontrollers or our framework on the SparkFun Edge board provides much better energy efficiency figures than running STM32Cube.AI or our framework on the Nucleo-L452RE-P board.

8. Conclusions

In this work, we presented a framework to perform quantization and then deployment of deep neural networks on microcontrollers. This framework represents an alternative to the STM32Cube.AI proprietary solution or TensorFlow Lite for Microcontrollers, an open-source but complex environment. Inference time and energy efficiency measured on two different embedded platforms demonstrated that our framework is a viable alternative to the aforementioned solutions to perform inference of deep neural networks. Our framework also introduces a fixed-point on 16-bit integers post-training quantization which is not available with the two other frameworks. We have shown that this 16-bit fixed-point quantization provides an improvement over a 32-bit floating-point inference, while being competitive with fixed-point on 8-bit integers quantization-aware training. It provides a reduced inference time compared to floating-point inference. Moreover, the memory footprint is dived by two while keeping the same accuracy. The 8-bit quantization provides further improvements in inference time and memory footprint but at the cost of a slight accuracy decrease and a more complex implementation.

Work is still in progress to implement some optimizations techniques for fixed-point on 8-bit integers inference. Three optimizations are especially targeted: per-filter quantization, asymmetric range and non-power of two scale factor. On top of that, using SIMD instructions in the inference engine should help further decreasing inference time. These optimizations would therefore make our framework more competitive in terms of inference time and accuracy. Another possible improvement for fixed-point on integers inference consists of using 8-bit quantization for weights and 16-bit quantization for activations. TensorFlow Lite for Microcontrollers is currently in the process of implementing this technique. Mixed precision can indeed provide a way to reduce the memory footprint of layers that do not need a high precision representation (using 8-bit for weights and activations), while keeping a higher precision (16-bit representation) for layers that need it. The CMix-NN [57] library already provides an implementation of convolution functions for various data types of configuration (in 2, 4 and 8 bits). To further improve power consumption and memory footprint, binary neural networks can also be considered. However, to run them efficiently on microcontrollers, binary neural networks need to be implemented using bit-wise operations on 32-bit registers. This way, as many as 32 computations could be performed in parallel.

Apart from quantization, other techniques can be used to improve the execution of deep neural network on embedded targets. One of these techniques is the big/LITTLE DNN approach [58] where the inference is done first on a very small deep neural network.

Then, if the confidence is too low, inference is done using a larger deep neural network to reduce the confusion of the classification task. This technique allows a fast inference response time for most inputs, thus lowering the power consumption. In fact, it has been shown that only a small set of inputs are difficult to classify and so require running the bigger deep neural network. However, this approach does not lower the memory footprint. Other techniques such as pruning can also be used to obtain a smaller deep neural network while keeping the same accuracy. When structured pruning [59] is used for instance, the entire filters are completely removed from the convolutional neural network model. This would reduce both the memory footprint and the power consumption. Finally, other optimizations techniques also consider new neural network architectures. One can cite for example the recently published MCUNet [2] framework with its TinyNAS tool that intends to identify a neural network model that performs well on the target.

Future works will also be dedicated to the deployment of neural network architectures on FPGA using High-Level Synthesis tool such as Vivado. In fact, a feasibility study has already been performed and has shown that our framework can be used for a deployment on FPGA as well. Moreover, work is in progress to natively support automatic PyTorch deployment. To do so, the features provided by the `torch.fx` module of the newly released PyTorch 1.8.0 is used.

Finally, we are currently working on a real application of our framework that consists in integrating artificial intelligence into smart glasses [60] to perform human activity recognition in the context of elder care among other tasks. Preliminar results have been published in [6].

Author Contributions: Investigation, P.-E.N.; methodology, P.-E.N. and G.B.H.; software, P.-E.N. and G.B.H.; supervision, A.P., B.M. and V.G.; writing—original draft preparation, P.-E.N.; writing—review and editing, G.B.H., A.P., B.M., V.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by "Université Côte d'Azur", "CNRS", "Région Sud Provence-Alpes-Côte d'Azur, France" and "Ellcie Healthy".

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. Comparison of Inference Time between a Microcontroller, a CPU and a GPU

Table A1. Microcontroller (STM32L452RE), CPU (Intel Core i7-8850H) and GPU (NVidia Quadro P2000M) platforms. Power consumption figures for GPU and CPU are the TDP values from the manufacturer and do not reflect the exact power consumption of the device.

Platform	Model	Framework	Power Consumption
MCU	STM32L452RE	STM32Cube.AI	0.016 W
CPU	Intel Core i7-8850H	TensorFlow	45 W
GPU	NVidia Quadro P2000M	TensorFlow	50 W

Table A2. Comparison of 32-bit floating-point inference time for a single input on a microcontroller, a CPU and a GPU. Neural network architecture is described in Section 6 with several filters per convolution layer varying from 16 to 80 and dataset is described in Section 6.1.1. For CPU and GPU, inference batch size is set to 512 and the dataset is repeated 104 times to try to compensate for the large startup overhead compared to the total inference time. Measurements are averaged over at least 5 runs.

Platform	Inference Time (ms)						
	16 Filters	24 Filters	32 Filters	40 Filters	48 Filters	64 Filters	80 Filters
MCU	85	174	271	404	544	921	1387
CPU	0.0396	0.0552	0.0720	0.0937	0.1134	0.1538	0.2046
GPU	0.0227	0.0197	0.0223	0.0284	0.0317	0.0395	0.0515

Appendix B. Comparison between TensorFlow Lite for Microcontrollers and MicroAI Quantizations

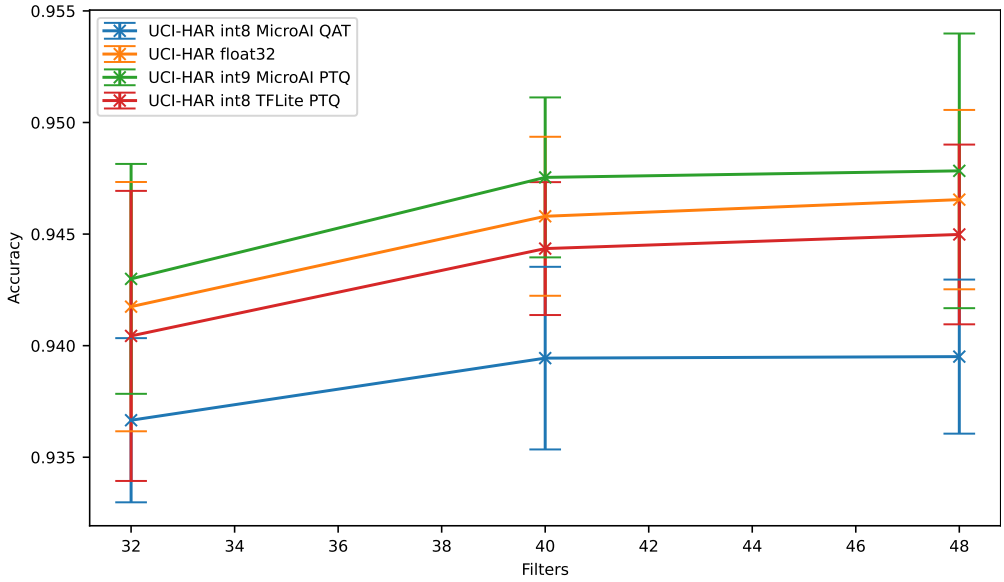

Figure A1. Accuracy vs. filters for baseline (*float32*), 8-bit Post-Training Quantization from TensorFlow Lite (*int8 TFLite PTQ*), 8-bit Quantization-Aware Training from our framework (*int8 MicroAI QAT*), and 9-bit Post-Training Quantization from our framework (*int9 MicroAI PTQ*). Neural network architecture is described in Section 6 with several filters per convolution layer varying from 32 to 48 and dataset is described in Section 6.1.1.

Appendix C. MicroAI Commands to Run for Automatic Training and Deployment of Deep Neural Networks

Data can be preprocessed (e.g., to apply normalization) from the source dataset and serialized to an intermediate dataset file with the following command:

```
microai <config.toml> preprocess_data
```

The training phase is started by running the following command:

```
microai <config.toml> train
```

Before being deployed and evaluated, the appropriate code must be generated and built for the targeted platform by running the following command:

```
microai <config.toml> prepare_deploy
```

Once the binaries are generated, they can be deployed, and the model can be evaluated on the target by running the following command:

```
microai <config.toml> deploy_and_evaluate
```

Appendix D. Detailed Results of Frameworks and Embedded Platforms Evaluation

Table A3. ROM footprint vs. filters for TFLite Micro, STM32Cube.AI and MicroAI.

			ROM Footprint (kiB)						
Framework	**Target**	**Data Type**	**16 Filters**	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
TFLiteMicro	SparkFunEdge	float32	116.520	133.988	157.957	188.426	225.395	318.926	438.363
MicroAI	SparkFunEdge	float32	54.316	67.066	91.035	121.512	158.473	251.863	371.332
MicroAI	NucleoL452REP	float32	55.770	68.145	92.129	122.582	159.559	253.004	372.434
STM32Cube.AI	NucleoL452REP	float32	61.965	79.449	103.410	133.898	170.859	264.289	383.742
MicroAI	SparkFunEdge	int16	46.952	50.629	62.629	77.832	96.355	142.973	202.699
MicroAI	NucleoL452REP	int16	48.129	51.629	63.613	78.855	97.340	144.051	203.770
TFLiteMicro	SparkFunEdge	int8	111.051	117.066	124.691	133.957	144.832	171.473	204.613
MicroAI	SparkFunEdge	int8	43.256	42.249	48.229	55.854	65.089	88.343	118.202
MicroAI	NucleoL452REP	int8	45.038	43.474	49.464	57.078	66.322	89.683	119.541
STM32Cube.AI	NucleoL452REP	int8	72.742	77.746	84.336	92.582	102.430	126.996	158.098

Table A4. Inference time for one input vs. filters for TFLite Micro, STM32Cube.AI and MicroAI.

			Response Time (ms)						
Framework	**Target**	**Data Type**	**16 Filters**	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
TFLiteMicro	SparkFunEdge	float32	179.633	294.157	438.541	624.172	860.835	1406.945	2087.241
MicroAI	SparkFunEdge	float32	53.247	153.732	259.212	394.494	569.852	1017.118	1561.264
MicroAI	NucleoL452REP	float32	55.762	152.426	259.160	395.721	559.249	976.732	1512.143
STM32Cube.AI	NucleoL452REP	float32	85.359	174.082	271.362	403.898	544.406	921.646	1387.083
MicroAI	SparkFunEdge	int16	40.867	113.035	191.439	287.655	389.450	667.547	1041.617
MicroAI	NucleoL452REP	int16	44.915	120.308	205.499	318.310	459.880	796.310	1223.513
TFLiteMicro	SparkFunEdge	int8	92.529	130.760	172.673	225.092	280.942	418.198	591.785
MicroAI	SparkFunEdge	int8	39.417	101.704	172.551	259.830	375.840	658.441	1003.365
MicroAI	NucleoL452REP	int8	43.003	107.705	180.830	272.986	383.761	659.996	1034.033
STM32Cube.AI	NucleoL452REP	int8	32.297	53.871	80.388	111.635	146.022	242.002	352.079

Table A5. Energy consumption for 1 input vs. filters for TFLite Micro, STM32Cube.AI and MicroAI.

			Energy (µWh)						
Framework	**Target**	**Data Type**	**16 Filters**	**24 Filters**	**32 Filters**	**40 Filters**	**48 Filters**	**64 Filters**	**80 Filters**
TFLiteMicro	SparkFunEdge	float32	0.135	0.221	0.330	0.469	0.647	1.058	1.569
MicroAI	SparkFunEdge	float32	0.040	0.116	0.195	0.297	0.428	0.765	1.174
MicroAI	NucleoL452REP	float32	0.247	0.675	1.148	1.753	2.478	4.327	6.700
STM32Cube.AI	NucleoL452REP	float32	0.378	0.771	1.202	1.789	2.412	4.083	6.146
MicroAI	SparkFunEdge	int16	0.031	0.085	0.144	0.216	0.293	0.502	0.783
MicroAI	NucleoL452REP	int16	0.199	0.533	0.910	1.410	2.038	3.528	5.421
TFLiteMicro	SparkFunEdge	int8	0.070	0.098	0.130	0.169	0.211	0.314	0.445
MicroAI	SparkFunEdge	int8	0.030	0.076	0.130	0.195	0.283	0.495	0.754
MicroAI	NucleoL452REP	int8	0.191	0.477	0.801	1.209	1.700	2.924	4.581
STM32Cube.AI	NucleoL452REP	int8	0.143	0.239	0.356	0.495	0.647	1.072	1.560

Appendix E. Number of Integer ALU Operations for a Fixed-Point Residual Neural Network

Table A6. Number of arithmetic and logic operations with fixed-point on integers inference for the main layers of a residual neural network with f the number of filters (output channels), s the number of input samples, c the number of input channels, k the kernel size, n the number of neurons and i the number of input layers to the residual Add layer. Conv1D is assumed to be without padding and with a stride of 1.

	MACC (1 Cycle)	**Add (1 Cycle)**	**Shift (1 Cycle)**	**Max/Saturate (2 Cycles)**
Conv1D	$f \times s \times c \times k$	N/A	$2 \times f \times s$	$f \times s$
ReLU	N/A	N/A	N/A	$c \times s$
Maxpool	N/A	N/A	N/A	$c \times s \times k$
Add	N/A	$s \times c \times (i-1)$	$s \times c \times i$	$c \times s$
FullyConnected	$n \times s$	N/A	$2 \times n$	n

References

1. Wang, Y.; Wei, G.; Brooks, D. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. *arXiv* **2019**, arXiv:1907.10701.
2. Lin, J.; Chen, W.M.; Lin, Y.; Cohn, J.; Gan, C.; Han, S. MCUNet: Tiny Deep Learning on IoT Devices. In Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS 2020), Online, 6–12 December 2020.
3. Lai, L.; Suda, N. Enabling Deep Learning at the IoT Edge. In Proceedings of the International Conference on Computer-Aided Design (ICCAD'18), San Diego, CA, USA, 5–8 November 2018; Association for Computing Machinery: New York, NY, USA, 2018. [CrossRef]
4. Kromes, R.; Russo, A.; Miramond, B.; Verdier, F. Energy consumption minimization on LoRaWAN sensor network by using an Artificial Neural Network based application. In Proceedings of the 2019 IEEE Sensors Applications Symposium (SAS), Sophia Antipolis, France, 11–13 March 2019; pp. 1–6. [CrossRef]
5. Tan, M.; Le, Q. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In Proceedings of the 36th International Conference on Machine Learning (PMLR 2019), Long Beach, CA, USA, 9–15 June 2019; Chaudhuri, K., Salakhutdinov, R., Eds.; Volume 97, pp. 6105–6114.
6. Novac, P.E.; Russo, A.; Miramond, B.; Pegatoquet, A.; Verdier, F.; Castagnetti, A. Toward unsupervised Human Activity Recognition on Microcontroller Units. In Proceedings of the 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 26–28 August 2020; pp. 542–550. [CrossRef]
7. Pimentel, J.J.; Bohnenstiehl, B.; Baas, B.M. Hybrid Hardware/Software Floating-Point Implementations for Optimized Area and Throughput Tradeoffs. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 100–113. [CrossRef]
8. Choi, J.; Chuang, P.I.J.; Wang, Z.; Venkataramani, S.; Srinivasan, V.; Gopalakrishnan, K. Bridging the accuracy gap for 2-bit quantized neural networks (qnn). *arXiv* **2018**, arXiv:1807.06964.

9. Esser, S.K.; McKinstry, J.L.; Bablani, D.; Appuswamy, R.; Modha, D.S. Learned step size quantization. *arXiv* **2019**, arXiv:1902.08153.
10. Nikolić, M.; Hacene, G.B.; Bannon, C.; Lascorz, A.D.; Courbariaux, M.; Bengio, Y.; Gripon, V.; Moshovos, A. Bitpruning: Learning bitlengths for aggressive and accurate quantization. *arXiv* **2020** arXiv:2002.03090.
11. Uhlich, S.; Mauch, L.; Yoshiyama, K.; Cardinaux, F.; Garcia, J.A.; Tiedemann, S.; Kemp, T.; Nakamura, A. Differentiable quantization of deep neural networks. *arXiv* **2019**, arXiv:1905.11452.
12. Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Binarized Neural Networks. In *Advances in Neural Information Processing Systems*; Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., Garnett, R., Eds.; Curran Associates, Inc.: Barcelona, Spain, 2016; Volume 29.
13. Rastegari, M.; Ordonez, V.; Redmon, J.; Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In Proceedings of the European Conference on Computer Vision, Amsterdam, The Netherlands, 8–16 October 2016; Springer: Cham, Switzerland, 2016; pp. 525–542.
14. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv* **2017**, arXiv:1704.04861.
15. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
16. Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255. [CrossRef]
17. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778. [CrossRef]
18. Han, S.; Pool, J.; Tran, J.; Dally, W. Learning both weights and connections for efficient neural network. In Proceedings of the 28th International Conference on Neural Information Processing Systems—Volume 1, Montreal, QC, Canada, 7–10 December 2015; MIT Press: Cambridge, MA, USA, 2015; pp. 1135–1143.
19. Yamamoto, K.; Maeno, K. PCAS: Pruning Channels with Attention Statistics. *arXiv* **2018**, arXiv:1806.05382.
20. Hacene, G.B.; Lassance, C.; Gripon, V.; Courbariaux, M.; Bengio, Y. Attention based pruning for shift networks. *arXiv* **2019**, arXiv:1905.12300.
21. Ramakrishnan, R.K.; Sari, E.; Nia, V.P. Differentiable Mask for Pruning Convolutional and Recurrent Networks. In Proceedings of the 2020 17th Conference on Computer and Robot Vision (CRV), Ottawa, ON, Canada, 13–15 May 2020; pp. 222–229.
22. He, Y.; Ding, Y.; Liu, P.; Zhu, L.; Zhang, H.; Yang, Y. Learning Filter Pruning Criteria for Deep Convolutional Neural Networks Acceleration. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Seattle, WA, USA, 14–19 June 2020; pp. 2009–2018.
23. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
24. Fard, M.M.; Thonet, T.; Gaussier, E. Deep k-means: Jointly clustering with k-means and learning representations. *Pattern Recognit. Lett.* **2020**, *138*, 185–192. [CrossRef]
25. Cardinaux, F.; Uhlich, S.; Yoshiyama, K.; García, J.A.; Mauch, L.; Tiedemann, S.; Kemp, T.; Nakamura, A. Iteratively training look-up tables for network quantization. *IEEE J. Sel. Top. Signal Process.* **2020**, *14*, 860–870. [CrossRef]
26. He, Z.; Fan, D. Simultaneously Optimizing Weight and Quantizer of Ternary Neural Network Using Truncated Gaussian Approximation. In Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Long Beach, CA, USA, 16–20 June 2019; pp. 11430–11438. [CrossRef]
27. Lee, E.; Hwang, Y. Layer-Wise Network Compression Using Gaussian Mixture Model. *Electronics* **2021**, *10*, 72. [CrossRef]
28. Vogel, S.; Raghunath, R.B.; Guntoro, A.; Van Laerhoven, K.; Ascheid, G. Bit-Shift-Based Accelerator for CNNs with Selectable Accuracy and Throughput. In Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 28–30 August 2019; pp. 663–667. [CrossRef]
29. Courbariaux, M.; Bengio, Y.; David, J.P. Training deep neural networks with low precision multiplications. *arXiv* **2015**. arXiv:1412.7024.
30. Holt, J.L.; Baker, T.E. Back propagation simulations using limited precision calculations. In Proceedings of the IJCNN-91-Seattle International Joint Conference on Neural Networks, Seattle, WA, USA, 8–12 July 1991; Volume ii, pp. 121–126. [CrossRef]
31. Vanhoucke, V.; Senior, A.; Mao, M.Z. Improving the speed of neural networks on CPUs. In Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NIPS 2011), Granada, Spain, 12–17 December 2011.
32. Garofalo, A.; Tagliavini, G.; Conti, F.; Rossi, D.; Benini, L. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, 9–13 March 2020; IEEE: New York, NY, USA, 2020; pp. 186–191. [CrossRef]
33. Cotton, N.J.; Wilamowski, B.M.; Dundar, G. A Neural Network Implementation on an Inexpensive Eight Bit Microcontroller. In Proceedings of the 2008 International Conference on Intelligent Engineering Systems, Miami, FL, USA, 25–29 February 2008; pp. 109–114. [CrossRef]
34. Nair, V.; Hinton, G.E. Rectified Linear Units Improve Restricted Boltzmann Machines. In Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10), Haifa, Israel, 21–24 June 2010; Omnipress: Madison, WI, USA, 2010; pp. 807–814.

35. Zhang, Y.; Suda, N.; Lai, L.; Chandra, V. Hello Edge: Keyword Spotting on Microcontrollers. *arXiv* **2018**, arXiv:1711.07128.
36. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*; IEEE: Piscataway, NJ, USA, 2019; pp. 1–84. [CrossRef]
37. Micikevicius, P.; Narang, S.; Alben, J.; Diamos, G.; Elsen, E.; Garcia, D.; Ginsburg, B.; Houston, M.; Kuchaiev, O.; Venkatesh, G.; et al. Mixed Precision Training. In Proceedings of the 6th International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
38. ARM. *ARM Developer Suite AXD and armsd Debuggers Guide, 4.7.9 Q-Format*; ARM DUI 0066D Version 1.2; Arm Ltd.: Cambridge, UK, 2001.
39. David, R.; Duke, J.; Jain, A.; Reddi, V.; Jeffries, N.; Li, J.; Kreeger, N.; Nappier, I.; Natraj, M.; Regev, S.; et al. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *arXiv* **2020**, arXiv:2010.08678.
40. STMicroelectronics. STM32Cube.AI. Available online: https://www.st.com/content/st_com/en/stm32-ann.html (accessed on 19 March 2021).
41. Google. TensorFlow Lite for Microcontrollers Supported Operations. Available online: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/kernels/micro_ops.h (accessed on 22 March 2021).
42. Google. TensorFlow Lite 8-Bit Quantization Specification. Available online: https://www.tensorflow.org/lite/performance/quantization_spec (accessed on 19 March 2021).
43. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 2704–2713. [CrossRef]
44. STMicroelectronics. Supported Deep Learning Toolboxes and Layers, Documentation Embedded in X-CUBE-AI Expansion Package 5.2.0, 2020. Available online: https://www.st.com/en/embedded-software/x-cube-ai.html (accessed on 19 March 2021).
45. Nordby, J. Emlearn: Machine Learning Inference Engine for Microcontrollers and Embedded Devices. 2019. Available online: https://doi.org/10.5281/zenodo.2589394 (accessed on 18 February 2021).
46. Sakr, F.; Bellotti, F.; Berta, R.; De Gloria, A. Machine Learning on Mainstream Microcontrollers. *Sensors* **2020**, *20*, 2638. [CrossRef]
47. Givargis, T. Gravity: An Artificial Neural Network Compiler for Embedded Applications. In Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC'21), Tokyo, Japan, 18–21 January 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 715–721. [CrossRef]
48. Wang, X.; Magno, M.; Cavigelli, L.; Benini, L. FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things. *IEEE Internet Things J.* **2020**, *7*, 4403–4417. [CrossRef]
49. Tom's Obvious Minimal Language. Available online: https://toml.io/ (accessed on 19 March 2021).
50. Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In Proceedings of the 32nd International Conference on Machine Learning, Lille, France, 6–11 July 2015; Bach, F., Blei, D., Eds.; PMLR: Lille, France, 2015; Volume 37, pp. 448–456.
51. Jinja2. Available online: https://palletsprojects.com/p/jinja/ (accessed on 19 March 2021).
52. Zhang, H.; Cissé, M.; Dauphin, Y.N.; Lopez-Paz, D. mixup: Beyond Empirical Risk Minimization. In Proceedings of the 6th International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
53. Davide, A.; Alessandro, G.; Luca, O.; Xavier, P.; Jorge, L.R.O. A Public Domain Dataset for Human Activity Recognition using Smartphones. In Proceedings of the ESANN, Bruges, Belgium, 24–26 April 2013.
54. Khacef, L.; Rodriguez, L.; Miramond, B. Written and Spoken Digits Database for Multimodal Learning. 2019. Available online: https://doi.org/10.5281/zenodo.3515935 (accessed on 18 February 2021).
55. Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv* **2018**, arXiv:1804.03209.
56. Stallkamp, J.; Schlipsing, M.; Salmen, J.; Igel, C. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In Proceedings of the 2011 International Joint Conference on Neural Networks, San Jose, CA, USA, 31 July–5 August 2011; pp. 1453–1460. [CrossRef]
57. Capotondi, A.; Rusci, M.; Fariselli, M.; Benini, L. CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices. *IEEE Trans. Circuits Syst. II Express Briefs* **2020**, *67*, 871–875. [CrossRef]
58. Park, E.; Kim, D.; Kim, S.; Kim, Y.; Kim, G.; Yoon, S.; Yoo, S. Big/little deep neural network for ultra low power inference. In Proceedings of the 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS), Amsterdam, The Netherlands, 4–9 October 2015; pp. 124–132. [CrossRef]
59. Anwar, S.; Hwang, K.; Sung, W. Structured Pruning of Deep Convolutional Neural Networks. *J. Emerg. Technol. Comput. Syst.* **2017**, *13*, 1–18. [CrossRef]
60. Arcaya-Jordan, A.; Pegatoquet, A.; Castagnetti, A. Smart Connected Glasses for Drowsiness Detection: A System-Level Modeling Approach. In Proceedings of the 2019 IEEE Sensors Applications Symposium (SAS), Sophia Antipolis, France, 11–13 March 2019; pp. 1–6. [CrossRef]