Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Compact Recurrent Neural Networks for Acoustic Event Detection on Low-Energy Low-Complexity Platforms

(Article begins on next page)

19 February 2025

Compact recurrent neural networks for acoustic event detection on low-energy low-complexity platforms

Gianmarco Cerutti[1], Rahul Prasad[2], Alessio Brutti[1], Elisabetta Farella[1]

[1] *ICT-irst Fondazione Bruno Kessler*, Trento, Italy

[2]*School of Information Science, Manipal Academy of Higher Education, Manipal*, Karnataka, India

{gcerutti, brutti, efarella}@fbk.eu, prassd25@gmail.com

Abstract—**Outdoor acoustic event detection is an exciting research field but challenged by the need for complex algorithms and deep learning techniques, typically requiring many computational, memory, and energy resources. These challenges discourage IoT implementations, where an efficient use of resources is required. However, current embedded technologies and microcontrollers have increased their capabilities without penalizing energy efficiency. This paper addresses the application of sound event detection at the very edge, by optimizing deep learning techniques on resource-constrained embedded platforms for the IoT. The contribution is two-fold: firstly, a two-stage student-teacher approach is presented to make state-of-the-art neural networks for sound event detection fit on current microcontrollers; secondly, we test our approach on an ARM Cortex M4, particularly focusing on issues related to 8-bits quantization. Our embedded implementation can achieve 68% accuracy in recognition on Urbansound8k, not far from state-of-the-art performance, with an inference time of 125 ms for each second of the audio stream, and power consumption of 5.5 mW in just 34.3 kB of RAM.**

I. INTRODUCTION

Internet of Things (IoT) applications require a large number of heterogeneous devices to be distributed in a certain environment. Each of them can potentially generate a large amount of data to be sent via wireless transmission, affecting the energy autonomy and lifetime of devices. In addition, privacy issues increase. One successful approach is distributing the computation at the edge, i.e. performing local pre-processing, but also advanced processing (e.g. machine learning, classification), directly on the wireless node, at "the thing" level [1], [2]. Thanks to recent improvements in embedded technology, computationally powerful microcontrollers with consumption in the range of mW enables Artificial Intelligence (AI) at the edge. This reduces the amount of data transmitted, avoiding flooding an enormous quantity of raw data at the cloud level, and the related power consumption.

Sound Event Detection (SED) is an example of IoT application where this approach can make a difference. In fact, since we are interested in events and not in raw data, a near-sensor processing approach is opportune. SED, as well as acoustic scene recognition, can benefit from understanding events locally where they happen both in terms of privacy [3] and reaction time, which can be kept in the range of ms.
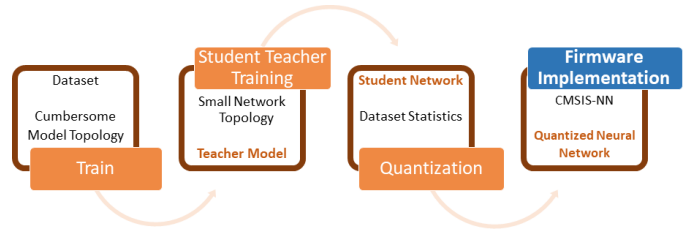

Fig. 1. From state-of-the-art to IoT: Starting from a sound event detection dataset and a state-of-the-art network topology that performs well on the problem, we apply the student-teacher approach to train a much smaller network that would fit on IoT devices. Finally, we quantize the compact network and write the firmware for low-energy platforms.

More than this, device lifetime can be guaranteed up to several years of operation, when energy harvesting is applied and the transmission is limited to few bytes. However, SED is a rather challenging task, especially when applied in outdoor contexts. After some pioneering efforts [4], [5], [6], which have not led to established solutions, recent progresses in deep learning and the release of sound event datasets and challenges like *UrbanSound8K* [7], AudioSet [8], ESC50 [9] and DCASE [10], [11] have reawakened interest in these applications, considerably improving the performance and paving the way to further developments. Nevertheless, advances in terms of accuracy and robustness of current acoustic event detection algorithms are achieved by using large neural networks, which are increasingly hungry in terms of computational power and memory. This prevents the development of applications for distributed monitoring in public spaces, which require a pervasive network of energy neutral devices composed of cheap, low-power, low-complexity platforms. An attractive way to reduce the network complexity while preserving as much as possible its generalization capabilities is through Knowledge Distillation (KD) [12]. Taking advantage of the redundancy that characterizes large networks [13], KD allows training small networks capable of mimicking large ones.

Going from state-of-the-art neural models to actual implementation on an IoT device involves multiple stages, as depicted in Fig 1. In our previous publication [14], we presented a KD approach to compress a SED classifier composed of the publicly available VGGish feature extractor [15] and

a recurrent classifier. Differently from common applications of KD, aimed at improving performance or at achieving limited reductions of the model dimensions, we obtained very high compression factors, reducing the network size from approximately 70 million parameters to nearly 20 thousand. These results paved the way to the effective use of deep neural networks on a low-power microcontroller to enable SED.

In this paper, we focus in particular on i) a preliminary analysis of the computational and memory requirements to understand what kind of models can be afforded by a given class of microcontrollers; ii) the quantization of the network parameters and the activations, presenting two different strategies to select for each layer the best fixed-point representation; iii) an implementation of the reduced network on a microcontroller with resources typical of an IoT end-node, building upon the network reduction strategies presented in [14] and iv) the evaluation of the accuracy of the actual implementation. In addition, we present an improvement of the KD approach presented in [14]. Distillation is performed in two stages where adaptation of the VGGish pre-trained feature extraction to the in-domain data is separated from the actual parameter distillation, leading to a further improvement of the classification accuracy.

II. RELATED WORKS

Nowadays, deep neural networks are state-of-the-art for many classification tasks, like image classification and speech recognition. The trend is to use networks of continuously increasing size and complexity because they generalize better than shallower ones [16]. As a consequence, the most recent and advanced solutions may be not practical if limited computational resources are available, as in the case of IoT application contexts. In fact, IoT nodes can be used as simple sensing devices, transmitting all the information directly to the cloud ([17], [18]). Nevertheless, this transmission is usually expensive from an energy point of view; thus, processing data locally on the node can be preferred, especially when the throughput is considerably high (e.g., audio, inertial movement unit, video). Therefore, in those IoT-related scenarios, reducing the network complexity and preserving as much as possible of its generalization capability is of significant interest. Fortunately, technology comes in to help, since the microcontrollers available nowadays are low-cost, energy-efficient processing units with average computation capability that allows non-trivial processing on board. Nevertheless, these systems still have some severe limitations, for example, in terms of memory limited to up to hundreds of kB).

Enabling advanced machine learning on IoT nodes is, therefore, of great interest and is becoming an attractive research topic for a variety of digital signal processing applications.

One way to achieve edge deep learning is by employing non-commercial platforms optimized for neural networks. As an example, the authors of [19] use a dedicated processing platform with state-of-the-art energy efficiency for an ultra-low-power deep-learning-powered autonomous nano-drone. In this way, no particular efforts are required in the network design thanks to the capabilities of the device. Processing

time and memory footprint are further reduced by properly quantizing the network. Quantization is also relevant in [20] where, in combination with the use of the CMSIS-NN library, an embedded-C framework for neural network developed for Cortex M4-M7 based microcontrollers [21], it allows a rapid and low-power classification of thermal images. However, the neural architecture is rather small (three-layer Convolutional Neural Network (CNN)), and the resolution of the thermal images is shallow (8x8).

In the previous examples, either the device has adequate resources or the feature dimensionality is very small; thus, a deep learning algorithm can run on the embedded platform. The improvement obtained via quantization is somehow "imposed" by the fixed point representation typical of an efficient microcontroller. However, for other classification tasks, in particular those involving the processing of audio streams, the approaches presented above are not practicable. The reason is that extensive neural networks, as well as feature vectors, are employed in state-of-the-art solutions.

Keyword spotting is another field of interest that requires always-on smart devices near-to-the speaker and thus calls for energy-efficient embedded systems with on-board recognition capabilities. A common strategy is to design and implement small networks, expressly fitted to the hardware capacity, that can be trained from scratch and implemented on low-power low-cost microcontrollers. [22] shows the superiority of CNN compared to fully connected deep neural networks in terms of performance, number of parameters and operations. Tang et al. implemented a set of residual neural networks with specific compact structures, focused on reducing the overall number of parameters and operations [23]. Zhang et al. implemented a keyword spotter on a commercial microcontroller using fixed-point quantization and a CMSIS-NN implementation [24], obtaining very short inference times.

A completely different strategy is to compress an existing model, generating a new network with a smaller memory footprint, but that effectively mimics the original one. In literature, several approaches exist to reduce the number of parameters of a neural network. Network pruning [25] aims at detecting and removing unimportant weights from a trained network until a given stop condition is reached. Matrix decomposition uses a compact format to represent the dense weight matrix of the fully-connected layers using few parameters, preserving the expressive power of the layer [26]. Matrix/tensor factorization [27], that exploits the linear structure of networks [28], and vector quantization of weights [29] are other strategies to reduce the network memory size. These methods reduce the amount of memory needed to accommodate the network (e.g., sharing the weights). However, they keep the same architecture, therefore requiring the same buffers (RAM) and throughput. Besides, network pruning requires a manual setup of sensitivity for each layer and fine-tuning of the parameters.

A further way to achieve model compression is to reduce the weight representation to very few bits. For example, BinaryConnect [30] and the related Binary Weight Net [31] represent weights with only 2 bits. If properly trained, the quantized networks can achieve performance close to the floating-point original models also on complex classification

tasks [32]. However, the memory and computational cost reductions are not sufficient for implementation on IoT devices where few KB are available (going from 32 to 2 bits results in a compression factor of 16 in the memory footprint). Experiments in [32], in fact, do not address the low-cost low-power devices we are targeting here. Additionally, non-conventional frameworks are needed to train the quantized network.

An attractive approach is to compress networks into a different and simpler architecture via KD [33]. This approach is also referred to as Student-Teacher because the smaller network (student) is trained to mimic the output of the larger one (teacher) [34]. The underlying idea is that the output of the neural network (soft labels) is more abundant in information than the hard labels and makes the training easier [12]. An example of network compression related to SED is [35]. Starting from the L^3 network for embedding extraction trained through self-supervised learning of audio-visual correspondence in videos [36], Kumari et al. compress this network targeting small edge devices, such as "motes" that use microcontrollers and achieve long-life self-powered operation. The work investigates the merging of different compression techniques (pruning, KD) and highlights the increase of performance using fine-tuning after compression.

Our proposed approach differentiates itself from those available in literature in multiple directions since it attempts to pull together the benefits of the methods reported above. We start from a large model and compress it. However, instead of just focusing on pruning or weight sharing, which provides limited memory reduction without decreasing the processing time, we design a minimal target network and use KD to train it (instead of training from scratch as in [22], [23], [24]). Note that distillation is typically employed to obtain limited network reductions while, in this paper, we target extremely high compression factors. On top of this heavy compression, we apply a stochastic weight and buffer quantization without the need for retraining the network.

The overview provided in this section shows the scientific interest in bringing intelligence to the edge in application domains such as computer vision and audio processing. Our work is positioned in this broad research field, then focusing on SED. To the best of our knowledge, this is the first attempt to use a student-teacher approach to perform sound event recognition directly on an IoT end-node in just 34.3 kB of RAM and 5.5 mW of power consumption.

III. KNOWLEDGE DISTILLATION

In this section, we give a brief overview of the Student-Teacher approach. A more detailed review is available in [14], where we present our proposed distillation strategy based on a compound loss function. In addition to this, here we introduce a two-stage distillation that provides a small but significant performance improvement.

Considering a generic architecture of a neural network, where the classifier follows a feature extractor, distillation can take place in different parts of the network. Figure 2 graphically shows the idea behind the distillation process. The
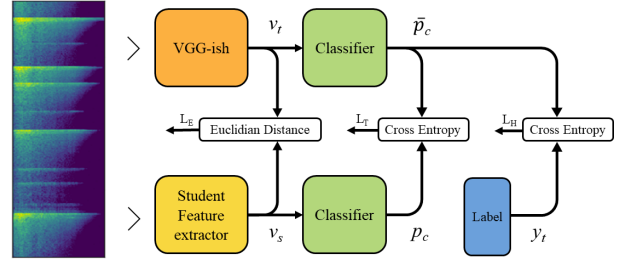

Fig. 2. Block diagram of KD using teacher intermediate features (L_E), soft teacher output (L_T) and dataset labels (L_H)

upper part represents the teacher network, the lower part is the student, and arrows indicate where the loss between the two networks is evaluated to train the student.

The original approach, [12], replaces the hard labels with the teacher output in the soft loss:

$$\text{Ł}_\text{s}(\mathbf{X}) = -\sum_{n=1}^{N} \sum_{c=1}^{C} \overline{p_c}(\mathbf{x}_n) \log(p_c(\mathbf{x}_n)), \tag{1}$$

where $\mathbf{X} = \{\mathbf{x}_1, \ldots \mathbf{x}_N\}$ is the set of input features, N is the number of samples, C is the number of classes, $c = \{1, \ldots, C\}$ is a generic class, $\overline{p_c}(\mathbf{x}_t)$ and $p_c(\mathbf{x}_t)$ are the logits of teacher and student respectively for input $\mathbf{x}_n$.

A further strategy, in line with [37], is to make the student learn how to replicate also the features produced by the teacher. The embedding loss $\text{Ł}_\text{e}(\mathbf{X})$ is thus defined as:

$$\text{Ł}_\text{e}(\mathbf{X}) = \sum_{n=1}^{N} ||v_t(\mathbf{x}_n) - v_s(\mathbf{x}_n)||^2, \tag{2}$$

where $v_t(\cdot)$ and $v_s(\cdot)$ are the feature vectors produced by teacher and student networks respectively.

In [14] we observed that the best solution is to combine different losses via a linear combination:

$$\mathcal{L}(\mathbf{X}) = \alpha_h \mathcal{L}_\text{h}(\mathbf{X}) + \alpha_s \mathcal{L}_\text{s}(\mathbf{X}) + \alpha_e \mathcal{L}_\text{e}(\mathbf{X}), \tag{3}$$

where $\text{Ł}_\text{h}(\mathbf{X})$ is the standard cross-entropy using the hard (one-hot) labels $\mathbf{Y} = \{\mathbf{y}_1, \ldots \mathbf{y}_N\}$:

$$\text{Ł}_\text{h}(\mathbf{X}) = -\sum_{n=1}^{N} \sum_{c=1}^{C} \mathbf{y}_n \log(p_c(\mathbf{x}_n)), \tag{4}$$

A. Dataset

For SED in outdoor urban environments, three datasets are often used in literature: *UrbanSound8K* [7], AudioSet [8], ESC50 [9] and TUT Sound events 2017 [10]. The latter is particularly attractive because it features real recordings and several comparative methods are available thanks to the related DCASE challenges [11]. Unfortunately, the task is very hard and the state-of-the-art accuracy is rather low. Moreover, the class distribution is highly unbalanced towards one class. Therefore the dataset does not allow a fair evaluation of KD methods. ESC50 is also rather in line with our application scenario, but its size is relatively small (3 minutes of audio per class) and does not allow generalizing the results. Finally, we also discarded AudioSet because its video-based labels, referring to scenes instead of isolated sound-events,

TABLE I
ARCHITECTURE OF THE MODELS UNDER ANALYSIS. "-" MEANS THAT THE
LAYER IS NOT ACTIVE, "X" MEANS THAT THE LAYER IS ACTIVE.

Layer	VGGish/M_{70M}	M_{20M}	M_{2M}	M_{200k}	M_{20k}
Conv1	64	64	32	8	4
Pool1	x	x	x	x	x
Conv2	128	128	64	16	8
Pool2	x	x	x	x	x
Conv3	256	256	128	32	16
Conv4	256	-	-	-	-
Pool3	x	x	x	x	x
Conv5	512	256	128	64	16
Conv6	512	-	-	-	-
Pool4	x	x	x	x	x
Conv7	-	-	-	64	32
Pool5	-	-	-	x	x
FC1	4096	2048	512	256	64
FC2	4096	2048	-	-	-
FC3	128	128	128	128	128
BatchNorm	x	x	x	x	x
GRU	20	20	20	20	20
FC4	10	10	10	10	10
#Param	~72.1M	~18.0M	~1.88M	~202k	~30.6k

Fig. 3. Average accuracy for the 4 different models in table I plus model M_{70M}. Each line represents a different training strategy. T_h is standard training, using just hard labels. $T_{h,s}$ uses both hard and soft labels. $T_{h,s,e}$ minimize also the feature vector produced by an intermediate level. Dashed line is the teacher accuracy, here referred as baseline.

require consistent additional work to be aligned with the label required for our analysis. Therefore, we focused on *UrbanSound8K*. It includes 8732 audio samples related to the city environment, with different sampling rates, number of channels and a maximum length of 4 seconds. Each recording has a unique label among 10 possible classes: air conditioner, car horn, children playing, dog bark, drilling, engine idling, gun shot, jackhammer, siren, and street music. All clips are taken from Freesound[1], a vast collaborative database of audio samples. Following the recipe reported in [7], we use 10-fold cross-validation and average scores. However, we took one additional fold for validation: 8 folds are used as training data, one is used as validation and the remaining one for test (training-validation-test ratio is 0.8-0.1-0.1). Validation fold is one index less than the test one (for example, when the test fold is 9, the validation fold is 8). Performance is measured in terms of classification accuracy. In all experiments, the dataset is augmented through pitch shifting [38], with both positive (up) and negative (down) semitones with values -2,-1,1,2.

B. Teacher

State-of-the-art solutions for SED are mostly CNN fed with mel-spectrogram [39][40][41]. However, to fully exploit the potential of the KD approach, rather than training from scratch our big CNN, we employed the publicly available VGGish[2] feature extractor [15], followed by a classification stage tailored on *UrbanSound8K*. VGGish is trained on the Youtube-8M Dataset [42] and it is expected to generalize well to other application contexts. Note that this fact introduces a further novelty in our work since distillation is performed on a different dataset than that used in the original training. VGGish converts 960 ms audio input mel spectrogram into a 128 dimensional embedding that can be used as input for a further classification model. The classifier can be shallow as the VGGish embeddings are more semantically representative
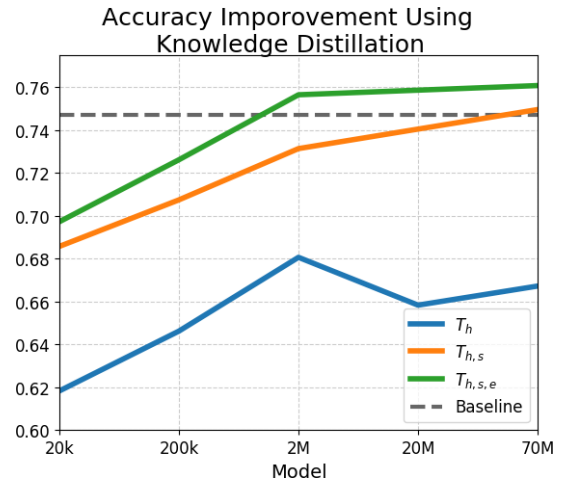
than raw audio features. The VGGish architecture is described in Table I. For all the convolutional layers the kernel size is 3, the stride is 1 and the activation function is ReLu. Max pooling layers are implemented with a 2x2 kernel and a stride of 2.

The classifier consists of a Gated Recurrent Unit (GRU) followed by a fully connected layers and maps the VGGish embeddings into the 10 classes of *UrbanSound8K*. A Batch-Normalization layer is inserted between the feature extractor and the classifier to accelerate training.

VGGish expects as input 960 ms of audio signal sampled at 16 kHz. Each clip in *UrbanSound8K* is resampled and padded or cropped to get a length of 960*4=3840 ms. The resulting signal is divided into 4 non-overlapping 960 ms frames. For each frame, the short-time Fourier transform is computed on 25 ms windows every 10 ms. The resulting spectrogram is integrated into 64 mel-spaced frequency bins, covering the range 125-7500 Hz, and log-transformed. This gives 4 patches of 96 x 64 bins that form the input of the VGGish.

C. Distillation

In this section, we analyze how the compression factor impacts on the final classification accuracy, using the standard loss and the loss compound described above. Although our goal is to fit the classifier on an IoT device, we consider four different degrees of compression, to better assess the potential and limits of the proposed approach, as reported in Table I. An heuristic adjustment of the layers and the number of filters is used to design the reduced networks. The model subscript M_x approximately represents the number of parameters of the upstream part. Note that the student networks drastically reduce the feature extractor only. The classifier is re-trained, but it keeps the same architecture, i.e. a 20-unit recurrent layer followed by a 10-unit fully connected layers.

Figure 3 reports the results in terms of classification accuracy for the 4 models and when training from scratch on *UrbanSound8K* and when distilling from the VGGish teacher.
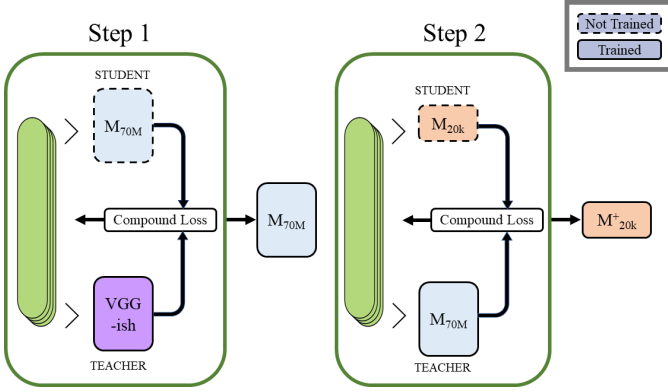
Fig. 4. Two stage distillation: M_{70M} is firstly trained from VGGish and the student M_{20k} is distilled from M_{70M}

In addition, we consider the model M_{70M}, which is a replica of the VGGish's architecture. All models considerably improve thanks to knowledge distillation with respect to being trained from scratch. Note also how the large models (M_{20M}, M_{70M}) perform much worse than VGGish and slightly worse than M_{2M} when trained from scratch: this is due to the fact that *UrbanSound8K* is not large enough for such huge networks. Finally, it is worth noting that the VGGish baseline is outperformed also by all models with more than 2M parameters. This is mostly due to the domain adaptation that we are implicitly applying when distilling knowledge from the teacher. As a matter of fact, the student feature extractor is tailored to the new in-domain data from *UrbanSound8K*, providing an improvement over the more general purpose VGGish feature extractor trained on Youtube8-M.

D. Two-stage distillation

The previous analysis, showing that we are jointly performing domain adaptation and parameter reduction, suggested us to investigate whether a further improvement can be achieved by separating these two processes.

Figure 4 describes our proposed 2-stage distillation, where we train the smallest network using the M_{70M} network as teacher. Note that M_{70M} is still trained using the compound loss in Eq. 3. Results are reported in Table II in comparison against training from scratch and using distillation from VGGish. The proposed approach achieves a 72.67% accuracy on the test set, providing a 3 points improvement over a more traditional distillation strategy (69.7%). Interestingly, the new M_{20k} model is just 2 points below the VGGish baseline and 4 points below M_{70M}, in spite of using less than 0.0424% of the parameter and 0.12% of the operation.

TABLE II
ACCURACY OF THE M_{20k} FOR: TRAINING FROM SCRATCH, DISTILLATION FROM VGG-ISH AND DISTILLATION FROM M_{70M}

	Standard Training	From VGG-ish	From M_{70M}
M_{20k}	61.83	69.72	72.67

IV. HARDWARE RESOURCES AND NETWORK REQUIREMENTS

A. Approximate hardware requirements per model

One interesting aspect towards tayloring our distilled networks for a resource constrained platform is to understand what are the computational and hardware requirements for a given model, or to understand what accuracy would be achievable with a given platform.

TABLE III
APPROXIMATE COMPUTATIONAL AND MEMORY REQUIREMENTS FOR EACH NETWORK

	VGGish	M_{20M}	M_{2M}	M_{200k}	M_{20k}
#Param	~72.1M	~18.0M	~1.88M	~202k	~30.6k
#Operations	~1.72G	~608M	~148M	~13.6M	~2.11M
#Buffer [B]	~614k	~602k	~301k	~76.0k	~34.3k

Table III reports an approximation of the computational requirements of each network: number of parameters to store, number of operations and buffer sizes. For memory requirements, we refer to an implementation using 8-bit quantized weights and buffers as specified in the CMSIS-NN library.

With an 8-bit quantization, each parameter takes one byte; therefore, the **non-volatile memory** in bytes needed by a model equals the total number of weights and biases (first row of Table III).

Buffers keep the outputs of each layer available during propagation and are stored in the **RAM**. The size of these buffers depends on the output dimensionality of each layer. However, the total amount of run-time memory required depends heavily on how efficiently buffers are implemented. The third row of Table III reports the RAM requirements of each model given the buffer design described in section V.

The **number of operations** (both multiplications and sums) depends on the layer type: in convolutional and maxpooling layers, each output pixel comes from a filter application. Given kernel size k and number of input channels c, each filter requires the following operations:

$$Ops_{filter} = 2 \cdot c \cdot k^2. \tag{5}$$

Therefore, the number of operations for each convolutional and maxpooling layer is:

$$Ops_{Conv} = Ops_{filter} \cdot out_H \cdot out_W \cdot out_C, \tag{6}$$

where out_H, out_W, and out_C are the output height, width an channel respectively.

For dense and GRU layers the number of operations is the number of matrix multiplications:

$$Ops_{DotProduct} = 2 \cdot in_{shape} \cdot out_{shape}. \tag{7}$$

Gated recurrent unit requires also three element-wise products.

B. Selection of the device class

Table IV shows a non-exhaustive list of processing platforms potentially adequate to be integrated into an end-device, with their power consumption and memory. In this section, we provide a qualitative analysis of what devices would be

able to run the distilled models presented in the previous sections. This analysis is inevitably rough as many figures are approximated and the actual values cannot be derived analytically. Additionally, we are not considering the time and resources required for other processing stages (e.g., Mel-bins extraction); thus, the constraints are rather relaxed with respect to the actual application. Nevertheless, the devices in Table IV show substantial differences in Million Instructions Per Second (MIPS) and memory (in the order of powers of 10); therefore, the results of this study are still valid as long as we refer to classes of devices.

Fig. 5. Models requirements vs hardware capabilities: a qualitative analysis Rectangles define network requirements : in terms of MIPS and RAM.

TABLE IV
EXAMPLES OF EMBEDDED PLATFORMS AND THEIR HARDWARE CAPABILITIES.

Board Name	Flash[KB]	RAM[KB]	Power [mW]	MIPS
Arduino	32	2	60	20
ChipKit uc32	512	32	181	124.8
STM32L476RG	1024	128	26	80
TI MSP432P4111	2048	256	23	58.56
BeagleBone Black	Ext	524288	2300	1607
Raspberry Pi 3 B+	Ext	1048576	5500	2800

Computational Cost. To ensure real time classification, the network must process each 960 ms audio frame (~1 second) before the next frame arrives. Thus, classification time must be shorter than 1 second. Unfortunately, converting exactly the Million Operation Per Second (MOPS) required by each model in the MIPS available on a given device, as reported in the datasheet, is not feasible. As a matter of facts, the number of instructions required by an operation depends on many factors, the most important being the actual implementation. In this analysis, we rely on the assumption that, on average, the number of operations is equal to the number of instructions. Typically, instructions are more than operations because each operation involves branch, load and store instructions. However, most of 32-bit microcontrollers support Single Instruction Multiple Data (SIMD), which allows up to four instructions in one clock cycle. We assume that these two effects are balanced and the number of operations is equal to the number of instructions. In the next sections, we will demonstrate that the actual throughput (operation per instruction) is larger than 1, but it gets close enough (1.8) in some situations. Given this assumption, in combination with the fact that other processing stages are not being accounted for, the MIPS available on the device must be larger (with some margin) than the MOPS needed in one forward propagation.

Memory constraints are also relevant: the RAM on-board must contain the intermediate values and the non-volatile memory should contain all network parameters. However, non-volatile memory is not the main limit in our examples: whenever the network does not fit in the flash memory it does not respect one of the others two parameters (RAM or MIPS).

Figure 5 roughly compares the devices in Table IV in terms of RAM and MIPS limitations against the compressed models each device can afford. None of the proposed models fits in the Arduino platform due to its limited RAM. The last two platforms of Table IV (BeagleBone Black and Raspberry Pi 3 B+) have enough RAM and MIPS to handle all models (actually VGGish fits only in Raspberrypi 3 B+). However,
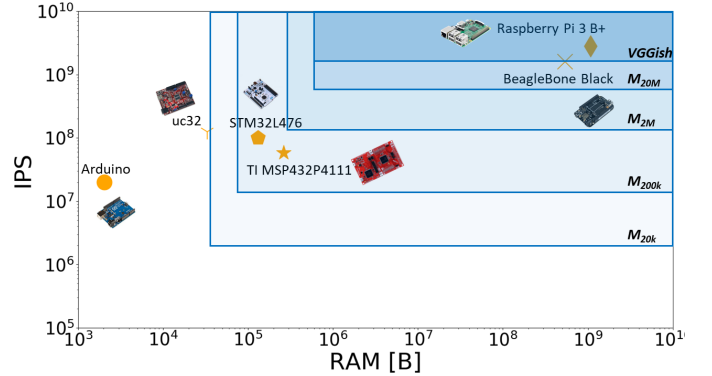
their power consumption is in the range of Watts that is not suitable for IoT applications. Therefore only devices ChipKit uc32, STM32L476RG, and TI MSP432P4111, which are approximately in the same class, are potentially suitable to host the distilled models. Uc32 is faster than both STM32 and TI platforms and would comfortably run the two smallest models in real-time, but its 32KB RAM is not large enough to fit even the ~$34.3\,\mathrm{kB}$ needed for the buffers of the smallest model M_{20k}. Concluding, M_{20k} and M_{200k} are the only suitable models for the target application and can be implemented in both the STM32L476RG and TI MSP432P4111 platforms. In the next section, we detail our implementation of M_{20k} in the STM32L476RG platform.

V. EMBEDDED PROGRAMMING

For the implementation of our SED system, we selected a Cortex M4 architecture operating up to $80\,\mathrm{MHz}$, which is a good compromise between efficiency in processing and flexibility in power management. In particular, we worked on an STM3276RG Nucleo board as a development platform. Thus, the reference values for current consumption are $420\,\mathrm{nA}$ in standby mode and $100\,\mu\mathrm{A/MHz}$ in full running mode. The Floating Point Unit (FPU) features single precision and implements a full set of optimized Digita Signal Processing (DSP) instructions for fixed-point operations.

Cortex-M4 provides SIMD instructions that operate on 8 or 16-bit integers. They are powerful for processing data such as video or audio, when full 32-bit precision is not necessarily required. The SIMD instructions allow 2 x 16 bit or 4 x 8 bit operations to be performed in parallel [43].

A. Quantization

Feed-forward propagation through a neural network requires vector/matrix/tensor multiplication and convolution. Therefore, all the core features employed in signal processing can be used for neural network computations. For this reason, ARM developed the CMSIS-NN framework for neural network propagation on top of DSP libraries [21]. The CMSIS-NN library maximizes performance and energy efficiency of common deep learning kernels on top of Cortex-M series cores.

Like for DSP, truncation of floating point numbers to 8 or 16-bit fixed point numbers improves the execution time and reduces the memory footprint. According to [21], 8-bit quantization achieves 4.6X improvement in runtime/throughput and 4.9X improvement in energy efficiency in image classification with CIFAR10 dataset. On the other hand, since quantization implies loss of precision, we could expect a direct impact on the final prediction performance. However, the authors of [44] experimented different kinds of quantization in image classification, achieving a 20% drop in model size without significant loss of accuracy. In the following, we describe the quantization process from a 32-bit floating-point to an 8-bit fixed-point representation. From now on, we will assume that floating point numbers has infinite precision and we will use the nomenclature typically used for DSP.

Quantization describes a real number with finite resolution. When a uniform quantization is applied, three parameters are used to define the fixed point representation: bit-width, step-size (resolution) and dynamic range [44]. These parameters are correlated by the following expression:

$$Range = Stepsize \cdot 2^{bitwidth-1}, \tag{8}$$

where $bitwidth - 1$ accounts for the bit used to represent the sign. $Stepsize$ is the minimum step between two fixed point numbers and will be always chosen as a power of two for convenience with binary representation.

An equivalent formulation of Eq. 8 can be obtained by considering the number of bits used for the decimal and integer parts of numbers:

$$Range = 2^{integer}$$
$$Stepsize = 2^{-decimal}$$
$$bitwidth = integer + decimal + 1$$

where, $integer$ is the number of bits used to represent the integer and $decimal$ is the number of bits used for the decimal part. +1 in the last equation accounts for the sign bit. Basically, given a fixed number $bitwidth$ of bits, increasing the range by increasing the number of bits for the integer part degrades the resolution. Conversely, decreasing $Stepsize$, by allocating more bits to the decimal part, leads to a range reduction.

The quantization error e_q is the difference between the infinite precision number and the quantized representation:

$$e_q = x - Q(x) \tag{9}$$

where x and $Q(x)$ are the input and the quantized output. If we treat the input as a random variable with a probability density function f_x (see [45]), the mean square quantization error is the combination of two different errors, namely granular error ($MSE_{q,g}$) and overload error ($MSE_{q,o}$):

$$MSE_{q,g} = \sum_{i=1}^{N} \int_{d_i}^{d_{i+1}} (x - Q(x))^2 f_x(x) dx$$

$$MSE_{q,o} = \int_{-\infty}^{d_0} (x - Q(x))^2 f_x(x) dx +$$
$$+ \int_{-\infty}^{d_N} (x - Q(x))^2 f_x(x) dx$$

where N is the number of quantization levels (in our case $2^{(bitwidth)}$) and d_i are the decision levels, i.e. any number between d_i and d_{i-1} is coded with the same fixed-point representation, usually $(d_i + d_{i-1})/2$. The two mean square errors are related to $Stepsize$ and $Range$ respectively. If $Stepsize$ is reduced then $MSE_{q,g}$ decreases; on the other hand if $Range$ is made wider, $MSE_{q,o}$ decreases.

The figure of merit linked with Mean Square Error (MSE) is the Signal to Quantization-Noise Ratio (SQNR), defined as:

$$SQNR = \frac{E[x^2]}{E[e_q^2]} = \frac{E[x^2]}{MSE_q} \tag{10}$$

Therefore, the goal is to define the optimum trade-off between $Range$ and $Stepsize$ to minimize the SQNR in the target application. In the following, we describe two different strategies to perform quantization using the quantization parameters and its figure of merit described above.

B. Quantization Design

Applying quantization to neural networks has different requirements with respect to other contexts, as image quantization or audio quantization. In particular, we are not interested in preserving an accurate representation of all activation outputs or weights for each layer. Conversely, we want that the final prediction is as close as possible to the prediction of the network with a 32-bit floating-point representation. To summarize, accuracy is the most relevant metric.

An exhaustive search of all the possible $integer/decimal$ combinations, evaluating the final accuracy, is not feasible in practice. In the smallest network, i.e. M_{20k}, the number of $integer/decimal$ digits would be 28, and each of them can vary between 0 and 7 (setting the decimal digit decides the integer part when $bitwidth$ is fixed). Therefore, the number of possible combinations is given by the permutation with repetition: $7^{28} = 4.5 * 10^{23}$.

The two solutions presented in this paper target the maximization of the SQNR or the reduction of the overload error.

The first approach to compute SQNR applies quantization on each variable (weights, activation) one at a time. For the weights, we choose the number of decimals that maximizes the SQNR. For the intermediate outputs we compute the SQNR running a forward propagation in floating-point on the training dataset and testing all the possible integer/decimal values (in the range from 0 to $bitwidth$). Each activation output is analyzed independently, eventually leading to different quantization trade-offs.

This approach relies on [44], where the overall SQNR, here defined as γ, is the harmonic mean of the SQNR of all preceding quantization steps:

$$\frac{1}{\gamma_{\text{output}}} = \frac{1}{\gamma_{a^{(0)}}} + \frac{1}{\gamma_{w^{(1)}}} + \frac{1}{\gamma_{a^{(1)}}} + \cdots + \frac{1}{\gamma_{w^{(L)}}} + \frac{1}{\gamma_{a^{(L)}}} \tag{11}$$

It turns out that maximizing the single SQNRs maximizes the overall SQNR, regardless of where quantization happens (at the first layers or at the last layers).

The second approach is based on the different effects of granular and overload errors. It is reasonable that a single

number with a high quantization error due to overload will affect the overall forwarding of the neural network. Conversely, the ensemble of small granular quantization error may not change the argmax of softmax layer, that is the value used to determine prediction and accuracy. Therefore, we select the integer/decimal ratio that reduces the probability of having values out of the quantization range. In particular, we set the number of bits for the integer part i such as

$$\min(i) : P(|x| < 2^i) < P_{th} \tag{12}$$

where x is the floating-point input and P_{th} is the probability of overload that one is willing to accept. This approach differs from the previous one for three reasons. (i) Numbers with large overload error have $bitwidth$ of the same importance than numbers with small overload error. (ii) Thus, each overload is considered uniformly, without accounting for the granular error but taking into only the overload error. (iii) This second approach has a hyper-parameter, i.e. the probability threshold, which requires a fine tuning on the training set. We heuristically set the threshold to 10^{-4}.

We estimate the probability density function $f(x)$ using the whole training set. The percentage of values out of the range is counted for all the possible decimal values involved in the network feed-forward propagation.

The upper part of Figure 5 is the histogram of the absolute value of the activation outputs between two consecutive powers of two. Most of the values (more than 10^6) are between -1 and 1 (or equivalently $|x| < 2^0$). Oppositely, just a small set of numbers (around 10^2) are such that $2^2 < |x| < 2^3$; they are depicted in the fourth bar. The central part of Figure 6 depicts the relative distrbution for each class of values, e.g. the estimated probability that a value is inside a given range related to two consecutive powers of 2. The values confirm what bars show: values in $|x| < 2^0$ are the most likely and values in $2^2 < |x| < 2^3$ appears only with a probability of 10^{-5}. Finally, the plot at the bottom depicts the SQNR using the number of bits for the decimal part available for a given range in the x axis (the range determines the bits for the integer part). For example, the most left point is the SQNR using 0 bits for integer part (which allows describing numbers in the range $0 \le |x| < 2^0$) and 7 bits for the decimal part. In this particular case, corresponding to a layer of the network, the optimum number of bits for the integer part obtained by maximizing the SQNR (three) does not correspond to what obtained by setting the probability threshold to 10^{-4} (four).

C. Firmware Programming

To transfer the neural model from to CMSIS-NN we developed an automatic python script to export the Keras model in an header file containing the weights to use in the firmware. To do so, firstly, we reordered the weights following the convention of both source and destination framework, then we quantized the weights using both quantization schemes. In this work, we set $bitwidth$ to 8 since we privilege execution time and power consumption with respect to accuracy. Note that we implemented a slightly different version of the M_{20k} model, replacing the GRU layer with a vanilla Recurrent
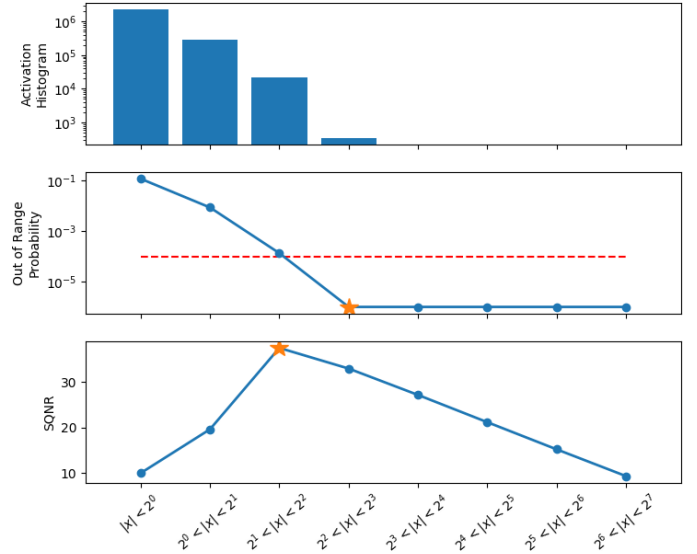

Fig. 6. The upper part shows a histogram of the activation values of one layer, between consecutive powers of 2. The lower part shows an example of the behaviour of the two quantization schemes investigated here. The central graph concludes that 3 decimals should be used for the integer part, because it is the first point in which probability goes below the threshold. The bottom line shows that two digits for the integer part are the solution with maximum SQNR and therefore the minimum MSE

Neural Network (RNN). The reason is that the implementation of a GRU layer in CMSIS-NN requires an additional effort while the accuracy does not change significantly.

The CMSIS-NN functions require different arguments: input dimension, number of channel and so on. These parameters should be extracted from the neural network and loaded in the microcontroller. Some of them requires extra processing, like shifting of weights and bias in internal operations. A detailed description of these parameters follows here.

Suppose that we are in an intermediate layer i and that the quantization procedure concludes that the number of integer bits m and the number of decimal bits n for weights, bias, input and output are m_w, n_w, m_w, m_b, m_i, n_i, m_o, m_o. The neural network operations include always a linear combination of weights and input, so that

$$output = input * weight + bias.$$

In multiplications between fixed point numbers, the number of decimal digits of the result are given by the sum of the number of decimal digits of the two operands. To sum the bias to this intermediate value, we need to use the same number of decimals. Thus, we shift the bias to make it match. In most of the cases it is a left shift. Finally, the output must have a certain number of decimals to get back in a fixed bitwidth format, so we apply a further shift. This last shift is opposite to the previous one and in CMSIS-NN is referred to as right shift. These concepts are expressed in formulas, implemented in the header files by means of a macro:

$$left\_shift = (n_i + n_w) - n_b$$
$$right\_shift = (n_i + n_w) - n_o$$

Another parameter to bear in mind is the size of the buffers. To necessarily instantiate buffers in the program, they need to

	I	C1	C2	C3	C4	C5
size$_{output}$	6144	**23312**	**10440**	4368	720	...
buffer	A	**B**	**A**	B	A	B

be known in advance. During inference, intermediate values are discarded, and only the final prediction and the network state (in case of recurrent neural network) are kept. As a consequence, it is possible to use the same buffers in multiple layers.

Each layer needs a pair of buffers able to contain output and input. To find the minimum size for these two buffers, we create a vector with the number of element between layers, $size_{output} = a_0, a_1, a_2, ...$, where a_0 is the input size. For each layer, the 2 buffers will switch their role: in the first convolutional layer, the input will be a_0 and the output a_1. For the next layer, a_1 is the input, a_2 is the output and so on. Table V shows the sequence of input and output for each layer in the implemented network. It shows that the two buffer sizes should be selected accounting for the maximum in odd and even indexes in the vector.

Finally, the CMSIS-NN framework requires also an additional small buffer for intermediate calculations, which can be reused in the implementation too.

VI. RESULTS

We evaluate the porting of our SED model to the microcontroller, in terms of power consumption, execution and recognition accuracy.

A. Accuracy

Input data from the test-set are sent by UART to the MicroController Unit (MCU) using a Python script. The forward propagation is computed inside the microcontroller that provides the prediction results on the same bus for accuracy evaluation. Following the recipe in [7], we used a 10-fold cross validation and average scores. For each test fold, we load models in the microcontroller with the quantization parameters computed on the related training set. Figure 7 depicts the accuracy over the 10 folds. The average accuracy shows a decrease of performance in both quantization schemes, but it is limited to a 2% drop overall if compared to the floating-point version (pink). Note that this minor performance drop could be limited by fine-tuning the quantized network [46].

Looking into more details, this performance deterioration is strongly dependent on the train/test-set split of each fold. Figure 8 reports the performance for three folds (train/test configurations): fold 7, fold 10, fold 2. When testing on fold 7, the performance deterioration is in line with the average accuracy decreases of around 2%. On the other hand, when fold 10 is selected as a test set, the accuracy drop is more consistent (8%). Finally, the impact of quantization error becomes negligible, or it even improves the classification results, in fold 2. This behaviour is related to the robustness of the original floating point model. As pointed out by Piczak [39],
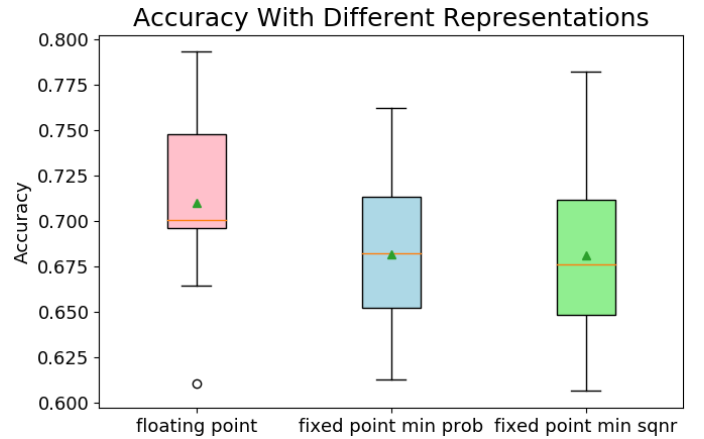

Fig. 7. Accuracy over the 10 folds for *Urbansound8K* dataset. Pink: floating point implementation. Blue: quantization by maximizing the SQNR. Green: quantization by minimizing the probability of overload error
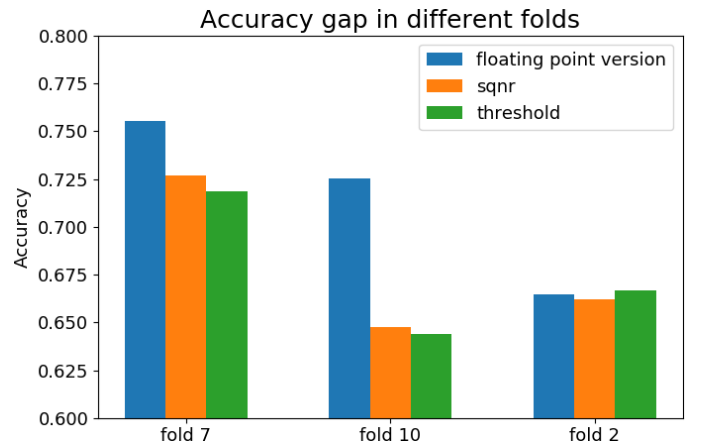

Fig. 8. Accuracy gap between floating point network and quantized versions for different folds. Accuracy degradation in fold 10 is more evident than in fold 2

some classes are more difficult to detect by a convolutional neural network, because of their short scale temporal structure (drilling, engine idling, jackhammer). Whenever the floating point network correctly classifies these classes, the difference between the likelihoods is probably shallow and the errors introduced by quantization can lead to a misclassification, leading to huge performance degradation. Similarly, the gap between the floating point and quantized network decreases or disappear when the reference network already does not classify correctly the difficult classes, as performance is already bad.

To confirm that, we compared the accuracy gap due to quantization and the F1 for these problematic classes. In all three cases, we confirmed that whenever the F1 score is high the accuracy drop increases, following an exponential profile as Figure 9 shows. The figure refers to the class "Jackhammer".

B. Execution Time and Power Consumption

In section III-A, we estimated the execution time of several network architectures by assuming that the number of operations is equivalent to the number of instructions, this way allowing a comparison between different platforms in terms
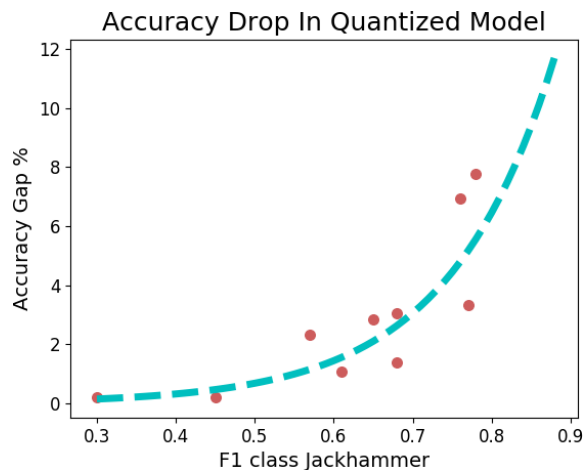
Fig. 9. Relation between performance in a specific class (Jackhammer) and the drop in accuracy due to quantization error. Due to its short-time duration, the class is difficult to detect for the convolutional model, thus its classification gets harder in the quantized model

of MIPS available from the datasheets. In this section, we evaluate the actual execution time and the power consumption of our implementation of the smallest model M_{20}. Measurements are done on the real prototype using the same system used to compute the accuracy. We measure the performance layer by layer to understand how each component contributes to the overall processing time. Table VI reports the evaluation results.

TABLE VI
EXECUTION TIME ON THE REAL PLATFORM, LAYER BY LAYER. IN THE BOTTOM, OVERALL EXECUTION TIME USING CMSIS-NN AND WITHOUT DSP OPTIMIZATION (PLAIN C)

Layer	Output Channel	#kop	Exec. Time [ms]	MOPS
Input$_{96,64}$	1	-	-	-
Conv1	4	419.61	63.93	6.56
Pool1	x	23.31	10.17	2.29
Conv2	8	751.68	20.13	37.34
Pool2	x	24.84	8.32	2.99
Conv3	16	628.99	12.90	48.76
Pool3	x	11.09	3.63	3.03
Conv4	16	207.36	3.88	53.44
Pool4	x	2.16	0.69	3.15
Conv5	32	27.65	0.60	46.23
Pool5	x	0.58	0.17	3.31
FC1	64	8.19	0.22	38,10
FC2	128	16.38	0.42	38.73
RNN	60	22.56	0.55	40,58
FC3	10	1.20	0.04	33.33
Total	-	2145.61	**125.6**	17.08
Plain C	-	2145.61	291.4	7.36

The CMSIS-NN framework implements a "basic" and a "fast" version of convolutional layers. The latter uses assembly directives to speed-up execution, especially by means of Multiply-ACcumulate (MAC) and SIMD. The only constraint for using this faster implementation is that the number of channels must be a multiple of 4 for 8-bit fixed-point quantization. This is the reason why, the number of channels of the convolutional layers of all architectures described in Table I are multiple of 4. Problems arise on the first layer, whose input is the Mel spectrum that has just one channel. This does not allow us to use the optimized version for convolutional layers and for this reason the first convolutional layer takes more

than half of overall execution time, with a throughput of 6.56 MOPS (see line 2 of Table VI). The second convolutional layer implements the fast version and takes just 20 ms, even if it is more computationally intense, performing 37.34 MOPS. This highlights how important is the parallelization and how much a proper parallelization in the first layer can reduce drastically the overall execution time.

To stress more the importance of an efficient framework for deep neural networks, we used a reference implementation of plain C (without explicit SIMD directives) for convolutional, maxpooling and fully connected layers. The comparison is in the last two lines of Table VI. The total execution time is speed-up of x2.32 with the fast implementation of CMSIS-NN with respect to plain C.

In Section IV, we stated that the combination of x4 parallelization and overhead, due to branches and load-store instructions, makes the microcontroller able to perform an 8-bit operation in just one instruction. The selected platform executes 80 MIPS, thus we expect 80 MOPS, but the overall throughput is different in real-time measurements (average 17 MOPS). Looking at the peaks in layer conv4, 53.44 MOPS is still far from our estimation of 80 MOPS. It means that parallelization does not fully compensate the overhead due to load-store and branches and we need approximately two instructions to execute an operation. The average (17.08 MOPS) is far from this peak level, mainly because of the first layer, that is the dominant part and it is not fully parallelized.

VII. CONCLUSIONS

In this work, we described the whole process from a state-of-the-art model for sound event detection to its energy efficient implementation in a microcontroller, targeting IoT applications. Firstly, we demonstrated that knowledge distillation can be effective also for extreme compression rates, achieving models suitable for real time applications on IoT nodes. Then, we introduced a two-step distillation to further improve the performance of the student network. Furthermore, we moved to the description of two quantization strategies, concluding that they perform in a similar way. Maximization of SQNR is generally preferred with respect to the probabilistic approach, because it does not require any hyperparameter. Both 8-bit quantization schemes were applied to the smallest distilled model resulting in a 2 percent points drop in accuracy, in comparison with the original floating point version. The final implementation on the microcontroller has a propagation time of 125 ms for each 1-second-audio-clip using just 5.5 mW average power and 34.3 kB of RAM. We have shown that an efficient framework for neural network, like CMSIS-NN, speed-up significantly the execution.

One interesting extension of our work would be to combine the distillation and quantization step, using the soft label to train the quantized networks. Finally, in future works we will implement the whole chain on an IoT node, including sensor acquisition, feature extraction and transmission of the classification outcome to the cloud.

REFERENCES

[1] F. Conti, R. Schilling et al., "An IoT endpoint system-on-chip for secure and energy-efficient near-sensor analytics," CAS, vol. 64, no. 9, pp. 2481–2494, Sep. 2017.

[2] M. Rusci, D. Rossi, E. Farella, and L. Benini, "A sub-mw IoT-endnode for always-on visual monitoring and smart triggering," IoT-J, vol. 4, no. 5, pp. 1284–1295, Oct 2017.

[3] P. Garcia Lopez, A. Montresor et al., "Edge-centric computing: Vision and challenges," SIGCOMM, vol. 45, no. 5, pp. 37–42, Sep. 2015.

[4] A. Temko, R. Malkin, C. Zieger, D. Macho, C. Nadeu, and M. Omologo, "CLEAR evaluation of acoustic event detection and classification systems," in CLEAR. Springer Berlin Heidelberg, 2007, pp. 311–322.

[5] X. Zhuang, X. Zhou, M. A. Hasegawa-Johnson, and T. S. Huang, "Real-world acoustic event detection," Pattern Recognition Letters, vol. 31, no. 12, pp. 1543 – 1551, 2010.

[6] A. Mesaros, T. Heittola, A. Eronen, and T. Virtanen, "Acoustic event detection in real life recordings," in European Signal Processing Conference, 2010, pp. 1267–1271.

[7] J. Salamon, C. Jacoby, and J. P. Bello, "A dataset and taxonomy for urban sound research," in MM. ACM, 2014, pp. 1041–1044.

[8] J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter, "Audio set: An ontology and human-labeled dataset for audio events," in ICASSP, March 2017, pp. 776–780.

[9] K. J. Piczak, "Esc: Dataset for environmental sound classification," in ACM international conference on Multimedia, 2015, pp. 1015–1018.

[10] A. Mesaros, T. Heittola, and T. Virtanen, "TUT database for acoustic scene classification and sound event detection," in EUSIPCO, 2016.

[11] A. Mesaros, T. Heittola, A. Diment, B. Elizalde, A. Shah, E. Vincent, B. Raj, and T. Virtanen, "DCASE 2017 challenge setup: Tasks, datasets and baseline system," in DCASE, 2017.

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

[13] Y. Le Cun, J. S. Denker, and S. A. Solla, "Advances in neural information processing systems 2," D. S. Touretzky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ch. Optimal Brain Damage, pp. 598–605.

[14] G. Cerutti, R. Prasad, A. Brutti, and E. Farella, "Neural network distillation on IoT platforms for sound event detection," in Interspeech, 2019.

[15] S. Hershey, S. Chaudhuri, D. P. Ellis et al., "CNN architectures for large-scale audio classification," in ICASSP, 2017, pp. 131–135.

[16] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," arXiv preprint arXiv:1605.07678, 2016.

[17] S. D. T. Kelly, N. K. Suryadevara, and S. C. Mukhopadhyay, "Towards the implementation of IoT for environmental condition monitoring in homes," IEEE sensors journal, vol. 13, no. 10, pp. 3846–3853, 2013.

[18] R. Piyare, A. L. Murphy, P. Tosato, and D. Brunelli, "Plug into a plant: Using a plant microbial fuel cell and a wake-up radio for an energy neutral sensing system," in IEEE 42nd LCN Workshops. IEEE, 2017, pp. 18–25.

[19] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "Ultra low power deep-learning-powered autonomous nano drones," in IEEE IROS, 2018.

[20] G. Cerutti, R. Prasad, and E. Farella, "Convolutional neural network on embedded platform for people presence detection in low resolution thermal images," in ICASSP, 2019, pp. 7610–7614.

[21] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm cortex-m cpus," arXiv preprint arXiv:1801.06601, 2018.

[22] S. Tara N. and P. Carolina, "Convolutional neural networks for small-footprint keyword spotting," in INTERSPEECH, 2015, pp. 1478–1482.

[23] T. Raphael and L. Jimmy, "Deep residual learning for small-footprint keyword spotting," in ICASSP, 2018, pp. 5484–5488.

[24] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," arXiv preprint arXiv:1711.07128, 2017.

[25] B. Hassibi, D. G. Stork, and G. J. Wolff, "Optimal brain surgeon and general network pruning," in IJCNN. IEEE, 1993, pp. 293–299.

[26] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing neural networks," in NIPS. MIT Press, 2015, pp. 442–450.

[27] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in NIPS, 2014, pp. 1269–1277.

[28] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in International Conference on Neural Information Processing Systems, 2013, pp. 2148–2156.

[29] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," Conference on Computer Vision and Pattern Recognition, pp. 4820–4828, 2016.

[30] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in NIPS, 2015, pp. 3123–3131.

[31] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in European Conference on Computer Vision, vol. 9908, 2016, pp. 525–542.

[32] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with ADMM," in Conference on Artificial Intelligence, 2018.

[33] C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model compression," in SIGKDD. ACM, 2006, pp. 535–541.

[34] J. Shen, N. Vesdapunt, V. N. Boddeti, and K. M. Kitani, "In teacher we trust: Learning compressed models for pedestrian detection," arXiv preprint arXiv:1612.00478, 2016.

[35] S. Kumari, D. Roy, M. Cartwright, J. P. Bello, and A. Arora, "EdgeL3: Compressing L3-Net for mote-scale urban noise monitoring," in IEEE International Parallel and Distributed Processing Symposium Workshops, 2019.

[36] J. Cramer, H.-H. Wu, J. Salamon, and J. P. Bello, "Look, listen, and learn more: Design choices for deep audio embeddings," in ICASSP, 2019, pp. 3852–3856.

[37] A. Romero, N. Ballas, S. E. Kahou, and Y. Bengio, "Fitnets: Hints for thin deep nets," arXiv preprint arXiv:1412.6550, 2014.

[38] J. Salamon and J. P. Bello, "Deep convolutional neural networks and data augmentation for environmental sound classification," IEEE Signal Processing Letters, vol. 24, no. 3, pp. 279–283, 2017.

[39] K. J. Piczak, "Environmental sound classification with convolutional neural networks," in MLSP. IEEE, 2015, pp. 1–6.

[40] Z. Zhang, S. Xu, S. Cao, and S. Zhang, "Deep convolutional neural network with mixup for environmental sound classification," in PRCV. Springer, 2018, pp. 356–367.

[41] X. Zhang, Y. Zou, and W. Shi, "Dilated convolution neural network with leakyrelu for environmental sound classification," in DSP. IEEE, 2017, pp. 1–5.

[42] J. F. Gemmeke, D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, and M. Ritter, "Audio set: An ontology and human-labeled dataset for audio events," in ICASSP. IEEE, 2017, pp. 776–780.

[43] T. Lorenser, "The dsp capabilities of arm cortex-m4 and cortex-m7 processors," ARM, Tech. Rep., November 2016.

[44] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in ICML, 2016, pp. 2849–2858.

[45] H. Sun and Y. Q. Shi, Image and video compression for multimedia engineering: Fundamentals, algorithms, and standards. CRC press, 2008.

[46] L. Zeng, Z. Wang, and X. Tian, "KCNN: Kernel-wise quantization to remarkably decrease multiplications in convolutional neural network," in International Joint Conference on Artificial Intelligence, 2019, pp. 4234–4242.