Arbitrary-Precision Convolutional Neural Networks on Low-Power IoT Processors

Arbitrary-Precision Convolutional Neural Networks on Low-Power IoT Processors

Valentino Peluso, Matteo Grimaldi, Andrea Calimera
Politecnico di Torino, 10129 Torino, Italy

***Abstract*—The deployment of Convolutional Neural Networks (CNNs) on resource-constrained IoT devices calls for accurate model re-sizing and optimization. Among the proposed compression strategies, n-ary fixed-point quantization has proven effective in reducing both computational effort and memory footprint with no (or limited) accuracy loss. However, its use requires custom components and special memory allocation strategies which are not available and burdensome to implement on low-power/low-cost cores. In order to bridge this gap, this work introduces *Virtual Quantization* (VQ), a hardware-friendly compression method which allows to implement equivalent n-ary CNNs on general purpose instruction-set architectures. The proposed VQ framework is validated for the IoT family of ARM MCUs (ARM Cortex-M) and tested with three different real-life applications (i.e. Image Classification, Keyword Spotting, Facial Expression Recognition).**

I. Introduction & Motivations

The success of the Internet-of-Things (IoT) lies under the ability to infer valuable information from the raw data gathered through distributed sensors. Also known as *sensemaking*, this process is commonly implemented as a cloud service where complex machine learning models, deep convolutional neural networks (CNNs) in particular, are run on high-performance computers. There is a wide consensus that a sensemaking stage implemented on the end-nodes is key to improve scalability of the IoT. Sensemaking on the edge is a means to ensure (i) real-time responses, (ii) lower volume of data and less energy consumption due to communication from/to the cloud, (iii) more data privacy [1]. The cloud-to-edge shift is not free-lunch, however. CNNs previously processed in the cloud with plenty of resources shall be processed in a mW power envelope using tiny processor cores with limited computational resources and low storage capacity. As an example, the RISC-based micro-controllers units (MCUs) designed by ARM for the IoT segment (i.e. the Cortex-M family reported in Table I) show limited integer arithmetic options (16- and 8-bit) and very small on-chip RAM (few hundreds of KB). This sets a clear limit to the complexity of CNNs that can be hosted.

Cortex-M	Power (μW/MHz)	RAM (KB)	Floating (32b)	Integer (16b,8b)	SIMD Unit (#lane)
M0	5.3	4-32	No	Yes	No
M3	11.0	32-128	No	Yes	No
M4	12.3	128-256	No/Optional	Yes	2
M7	33.0	256-512	No/Optional	Yes	2

Table I: Main features of the Cortex-M IoT MCUs by ARM [2]

Let's consider the three applications used as test-cases in this work, i.e. Image Classification (IC) on CIFAR-10 [3], Keyword Spotting (KWS) [4] and Facial Expression Recognition (FER) [5]. They are built upon the most lightweight Convolutional Neural Networks, nonetheless, their models do not fit into MCUs. Firstly, weights are trained using a 32-bit Floating-Point precision which is available for a few custom versions of Cortex M-4 and -7. When squeezed to 16-bit fixed-point, the models are still too large: IC (262 KB) would match the M7 core, which is the largest and more power hungry of the family, while for KWS (608 KB) and FER (1308 KB) the only option is to move the data off-chip. The use of external memory supports affects several metrics negatively, such as energy consumption, endurance, and reliability, integration cost. To notice that FER exceeds the RAM available in the M7 even with 8-bit (654 KB). A quantization below the 8-bit mark, e.g. 7-, 6-, 5-, 4- [6], till the extreme case of ternary [7] and binary [8] quantization, is a valuable option here.

Different implementations of arbitrary-precision quantization are currently available: dynamic compression schemes that play at different levels of granularity, e.g. layer-by-layer [9], [10], with different quantization rules, e.g. linear/non-linear and symmetric/asymmetric [11], or different radix point scaling factor, e.g. binary. When applied to over-parametrized CNN models, like AlexNet, VGG, ResNet, they achieve a memory compression which is proportional to the reduction of the bit-width with low accuracy loss [12]. Even if rarely tested on compact CNNs designed for embedded applications, like those targeted by this work, it is fair to assume they can achieve similar results. Unfortunately, this is just theory. As previously discussed, general purpose MCUs support a discrete set of integer options, e.g. 16-bit and 8-bit for the Cortex-M. Moreover, only very few basic fixed-point scaling schemes can be implemented with low performance overhead, i.e. linear. Some IoT platforms offer the 4-bit, e.g. the GAP-8 powered by the PULP core [13], but to the best of our knowledge, there are no ready-to-use IoT solutions for arbitrary bit-width integers. The Imagination Series 2NNX comes with a multi-bit instruction set, yet with a power budget of few Watts. For general purpose architectures, the storage of weights with arbitrary bit-widths needs custom memory allocation strategies to store multiple operands in a single memory word. As reported in [14], this may lead to huge performance penalty due to additional operations to unpack the operands and feed the execution units in a regular manner. To notice that custom hardware solutions developed on programmable devices (e.g. FPGAs [15]) are a too costly option for IoT applications.

The preliminary conclusion is that generic n-ary quantization schemes that do not take into consideration the actual hardware

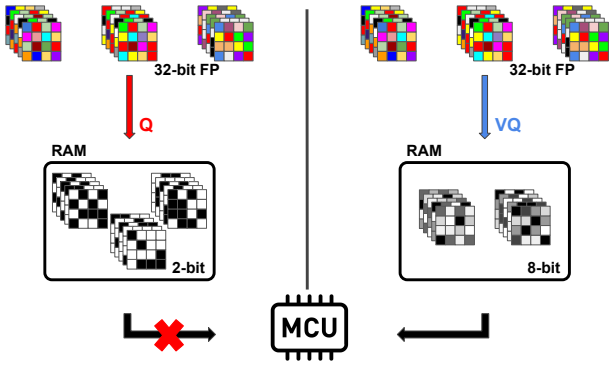

Figure 1: Standard n-ary Quantization (left) and the proposed Virtual Quantization (right); colors depths indicate the bit-width.

specifications merely fail on real life. This motivates our work which introduces *Virtual Quantization* (**VQ**), a software-level CNN compression method that allows to emulate the availability of n-bit instruction-set and hence the deployment of n-ary quantized models on general purpose IoT MCUs. Given the memory model of the target architecture and the arithmetic precision of its instruction set (8- and 16-bit for Cortex-M), **VQ** delivers a CNN with the same RAM footprint of the theoretic n-ary quantized model yet achieving better performance. The picture in Fig. 1 gives an overview of the main difference between a standard n-ary quantization **Q** and the proposed **VQ** when a full precision (FP) 32-bit CNN model has to be deployed on a MCU. With **Q** the maximum bit-width that allows to meet the RAM constraint is n=2; with **VQ** the bit-with is kept to n=8, the value supported by the instruction set, while the memory constraint is met by pruning some filters. Both **Q** and **VQ** satisfy the memory constraint, but only **VQ** can be run on-chip efficiently. Results collected on three applications (IC, KWS, FER) processed on the ARM Cortex-M cores demonstrate and quantify the savings brought by **VQ**.

II. Related Works

The urgent need of embedded CNNs that can run at the edge, on resource-constrained devices, accelerated the research studies on theories, methods, and tools for model compression. The two most adopted strategies, also adopted in **VQ**, are *quantization* and *pruning*. The next two paragraphs give a synthetic taxonomy of existing techniques, with emphasis on the most effective solutions applied to CNNs.

Quantization. Preliminary works demonstrated that 32-bit Floating-Point CNNs can be quantized to 16-bit and 8-bit Fixed-Point [12] ensuring minimal accuracy loss. Then, more extreme optimization techniques tried to further decrease the model precision using ternary [7] or binary [8] representations. Regardless the bit-width adopted, quantization can be defined as *fixed* if equally applied to all the layers, or *variable* when it uses different bit-width across the layers [6]. The weights can be quantized *linearly* or *non-linearly*. A *linear* approach [16] applies a uniform distance between all the quantized weights; this is the trivial solution, yet the most suitable for general-purpose hardware due to its lightweight implementation. The quantization range can be considered *symmetric*, if centered around zero, or *asymmetric* if shifted by a given offset; the choice is driven depending on the actual shape of the weights distribution. Finally, it is possible to quantize the weights using a binary radix point scaling or with an arbitrary linear scaling; the former allows the use of simple shift operations for scaling among the quantized levels [17], the later might be more accurate but it requires additional operations and hence more latency [17]. A *non-linear* quantization applies a particular function for mapping the full precision parameters into a discrete fixed-point space. It ensures a more accurate profiling of the original distribution, usually not uniform, guaranteeing lower accuracy losses. The most popular examples are the *log-domain* [18] and *clustering* approaches [19]. To notice that non-linear schemes imply the use of hashing functions which can be efficiently processed on custom hardware accelerators, but pose severe overhead when implemented as software routines.

Pruning. A pre-trained CNN may show irrelevant or less significant parameters that do not contribute to the inference accuracy, but rather they induce noise and redundant complexity. For such reason, they are good candidates to be removed. Pruning is mainly implemented as an accuracy-driven strategy, even though recent works introduced other extra-functional metrics in the optimization loop, e.g. energy [20]. The pruning stage can be implemented at different levels of granularity. The *Weights-level* is the finer granularity [19] where every single weight can be zeroed, both from fully-connected layers and convolutional layers. This strategy increases the network's sparsity, which is the ratio between zero and non-zero parameters. Hence, the inference task can be accelerated through a *sparse-matrix* multiplication. The latter is hard to be implemented onto low-power MCUs, where hardware resources are very constrained. A *Kernel-level* pruning works at a middle granularity [21]; weights are pruned as regular bunches [22], usually the 2-D arrays forming the 3-D filters of a convolutional layer, such that the utilization (i.e. parallelism) of the SIMD unit is maximized. Processing cores which lack a SIMD unit do not benefit much, thereby incurring the same limitations of weight-level methods. Finally, at the higher level of granularity, there is the so-called *filter pruning*, where the grain is an entire 3-D filter [23]. This is the most aggressive approach as it applies a drastic reshaping of the network topology which might affect accuracy dramatically. Nevertheless, it is the most effective strategy in terms of model compression and it does converge very quickly to a given memory bound.

As reported in the literature, the consensus is that fine-grain techniques are hard to fit into general purpose cores as they require special weights encoding, whereas coarse-grain approaches enable a straightforward implementation as they use a regular memory management and standard routines [22].

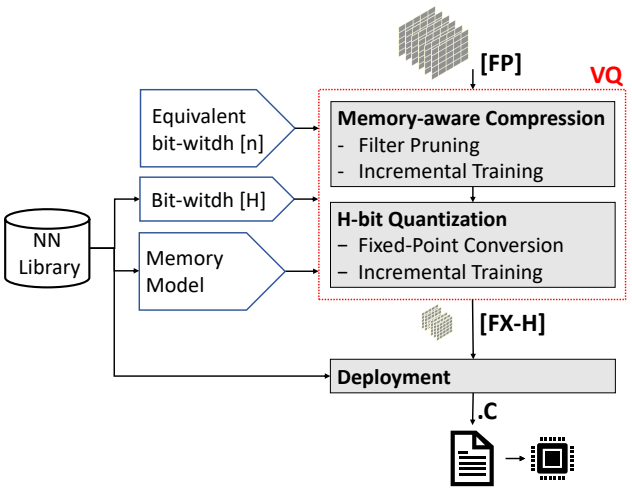

Figure 2: The Virtual Quantization flow.

III. Virtual Quantization

The aim of **VQ** is to devise a CNN model tuned for a target bit-width H s.t.: (i) the resulting memory footprint is the same that would have been obtained through a classical n-bit quantization, being $n < H$; (ii) the classification accuracy is larger or equal than that of the theoretic n-bit model. Given H as the bit-width supported by the instruction set of the target core, the model obtained with **VQ** does emulate the n-ary model while preserving hardware compliance.

A. Flow Overview

VQ is implemented through the optimization flow depicted in Fig. 2. The framework is fed with a floating-point CNN model (FP) trained with any standard deep-learning library (e.g. PyTorch, TensorFlow); it generates a `C` description of the equivalent n-bit quantized fixed-point model (FX-H), with n the equivalent bit-width provided as input. The high-level description of the network is translated into a low-level code using a neural network library (NN library) optimized for the target hardware (deployment stage in Fig. 2). In this work, we adopted the CMSIS-NN by ARM for the Cortex-M architecture [17]. The CMSIS-NN is a collection of optimized neural kernels which cover the most common operators: convolutions, fully-connected, activation and pooling functions. The final output is a `C` file that can be compiled and flashed on the target MCU. A memory model file describes the memory allocation strategy used to estimate the physical RAM needed at inference-time. It is worth emphasizing that the RAM usage drives the compression; this is a distinctive feature w.r.t. classical accuracy-driven compression methods.

The core of **VQ** consists of two main stages: (i) memory-aware compression; (ii) H-bit model quantization, with H as the bit-width supported by the instruction-set of the target core. As long as both the memory model and the neural network library are available, the **VQ** flow does apply to any micro-controller. Since the target of our work is the ARM Cortex-M architecture, we built a memory model for the CMSIS-NN library and we assume $H = 8$. The next sections give a detailed overview of the **VQ** steps, with a preliminary description of the memory allocation model.

B. Memory Model

Typical state-of-the-art CNNs are directed acyclic graphs whose scheduling is input-independent, hence the memory allocation can be done statically. One can precisely compute the memory footprint just knowing the network topology and the linked neural library (the CMSIS-NN by ARM in this work). The data structures needed for the feed-forward execution of a neural networks include: (i) a buffer storing the network parameters (weights and biases); (ii) a buffer for the input and output features; (iii) a buffer storing partial data used by neural networks routines. Since low-power MCUs, e.g. Cortex-M, have a very simple memory hierarchy, all the three buffers reside in RAM. The overall RAM space is thereby computed as the sum of the three contributions [17]. For what concerns the CNN parameters (i), they are permanently stored in the flash memory and then block-loaded in RAM at run-time thus to avoid the overhead of accessing the flash memory. Regarding the feature (ii) and internal (iii) buffers, the CNN layers are executed sequentially, therefore the corresponding memory can be time-shared between different layers.

The memory model is obtained from the GNU linker [24] and cross-validated with the statistics collected at run-time (we installed the lightweight mbed-os operating system for this purpose, v. 5.9.7). To estimate the memory footprint of the theoretic n-bit model, we extended the embedded memory model to arbitrary bit-widths assuming an ideal word size equal to n. To notice that this a theoretic model as in real hardware the minimum word is usually greater, i.e. $H > n$; for instance, $H = 8$ in the Cortex-M architecture.

C. Memory-aware Compression

A CNN with a fixed number of layers has two potential sources of redundancy: (i) the number of parameters within each layer; (ii) the arithmetic precision of the weights within each filter. The key observation over which **VQ** is built is that standard n-bit quantization operates on the second term only. Meeting a tight memory constraint would, therefore, require a bit-width n too small (usually much smaller than the minimum bit-width H made available by common HW). To overcome this issue, **VQ** implements a layer-wise compression which is based on a memory-aware filter pruning.

The iterative procedure of the memory-aware compression is described in the pseudo-code of Algorithm 1. At each iteration, the least important filter from the least important layer (lines 4–8) is dropped. As a *ranking* criterion, we used the sum of the absolute weights, i.e. the $\ell 1$-norm of the parameters. Weights with lower $\ell 1$-norm have less impact on the output features [23]. The loop iterates until the memory constraint is met (line 3). The memory estimation is run using the memory model introduced in the previous section (lines 1–2). As already discussed, it accounts for all the data structures used at the inference stage, not only the network parameters.

Algorithm 1: Memory-aware compression algorithm

Input: FP-Model, Bit-width H, Equivalent bit-width n
Output: Compressed Model

```
1  Target Memory = Memory of FP-Model at n-bits)
2  Current Memory = Memory of FP-Model at H-bits)
3  while Current Memory > Target Memory do
4      Layer ranking
5      Pick less important layer
6      Filter ranking
7      Remove less important fitler
8      Update Current Memory
9  Incremental training
10 return Compressed Model
```

It is worth emphasizing that the memory footprint is estimated depending on the physical bit-width of the target hardware, i.e. H, but the model is not quantized yet at this stage. This gives to the compression stage the proper awareness of quantization. The model compression might degrade the quality of results. To recover the accuracy loss, we leveraged an incremental training procedure (line 9).

D. H-bit Quantization

After memory-aware compression via pruning, the model undergoes the actual quantization to H-bits. The CMSIS-NN library offers fixed-point neural kernels with a per-layer dynamic scheme based on power-of-two scaling. Adopting a per-layer radix-point brings to better results as different layers show different dynamic ranges. Even though more complex scaling techniques, e.g. asymmetric quantization, might result more accurate, they might increase the execution time when the CNN is deployed on low-power MCUs, up to 20% according to [17]. The accuracy drop induced by quantization can be recovered (totally or partially depending on the actual constraints) using an incremental re-training procedure. The latter has the following main characteristics: the forward-propagation is run with fixed-point emulation; during back-propagation weights are kept in a floating-point format thus to allow small weight updates; weights are quantized at every epoch using stochastic rounding. In order to emulate fixed-point arithmetic on GP-GPUs we also implemented an in-house tool that leverages the fake-quantization method introduced in [25]. It consists of a software wrapper that converts activations and weights (stored in fixed-point) to the 32-bit floating-point; after being processed, results are converted back to fixed-point.

IV. Experimental Results

A. Benchmarks, Datasets and Training

We tested the **VQ** framework on three popular tasks: Image Classification (IC), Keyword Spotting (KWS), Facial Expression Recognition (FER). Different lightweight CNNs suited for tiny cores are deployed for each task; details reported in Table II. Such CNNs are trained for 150-epochs in PyTorch (version 0.4.1) using Adam optimization [26] (learning rate $1e-3$, linear decay 0.1 every 50-epochs, batch-size 128).

Table II: Benchmark models architecture. Considering that each convolutional layer with shape (c_{out}, f_y, f_x), fully-connected with shape (c_{out}), and max-pooling layer with shape (f_y, f_x). f_y and f_x indicate respectively the height and width of input features, while c_{out} is the number of output channels.

IC		**KWS**		**FER**	
CIFAR-10 [3]		Speech Commands [4]		FER2013 [5]	
Input: $3 \times 32 \times 32$		Input: $1 \times 32 \times 40$		Input: $1 \times 48 \times 48$	
Conv2d	(32,5,5)	Conv2d	(64,20,8)	Conv2d	(32,3,3)
MaxPool2d	(3,3)	MaxPool2d	(1,3)	Conv2d	(32,3,3)
Conv2d	(32,5,5)	Conv2d	(64,10,4)	Conv2d	(32,3,3)
MaxPool2d	(3,3)	MaxPool2d	(1,1)	MaxPool2d	(2,2)
Conv2d	(64,5,5)	FC	(32)	Conv2d	(64,3,3)
MaxPool2d	(3,3)	FC	(128)	Conv2d	(64,3,3)
FC	(10)	FC	(12)	Conv2d	(64,3,3)
				MaxPool2d	(2,2)
				Conv2d	(128,3,3)
				Conv2d	(128,3,3)
				Conv2d	(128,3,3)
				MaxPool2d	(2,2)
				FC	(7)

For **IC** we used the CNN delivered within the Caffe framework [27] according to the experimental set-up reported in [17]. The data-set is the popular *CIFAR-10* [3], made up of 60000 32×32 RGB images labeled with 10-classes. Concerning **KWS**, we followed the experimental procedure introduced in [28], which makes use of the *cnn-trad-fpool3* CNN [28] to classify 10 keywords belonging to the Speech Command Dataset [4]. The training-set and test-set data are composed of 56196 and 7518 spectrograms respectively ($time \times frequency = 32 \times 40$). For the **FER** task we resorted to a VGG-like CNN which recognizes the facial emotion dataset provided by [5]. The dataset has 48×48 grayscale facial images classified by 7 labels; training and test set consist of 28708 and 3589 instances respectively.

B. Deployment and Emulation

We validated the **VQ** framework (Fig. 2) for the Cortex-M family by ARM. Tests were run on the NUCLEO-F767ZI board by ST Microelectronics using the CMSIS-NN library v.5.4.0 provided by ARM. The GNU Arm Embedded toolchain (version 7.3.1) was used to compile the `C` level model. In order to emulate the n-bit quantization (for which there's no HW available) and to ensure a fair comparison with the **VQ** method, the inference accuracy of the three applications is measured through the fake-quantization tool mentioned in Section III-D. Such a tool is made run on a GP-GPU workstation powered with a Titan GTX-1080 Ti by NVIDIA and it offers several settings that adapt to different fixed-point HW units. For what concerns this work, the tool is tuned for the ARM Cortex-M integer unit. Extensive emulation runs show fake-quantization achieves 100% match with the results computed on the actual boards.

C. Results

Table III collects the results obtained with a standard n-bit quantization (**Q**) where models are scaled to an arbitrary fixed-point (FX) bit-width n. For a fair comparison, the adopted **Q**

Table III: Performance of n-ary quantization **Q**.

Task	D-Type	n-bit	RAM (KB)	Core	Top-1 (%)
IC	FP	32	n/n	None	82.80
	FX	8	131	M4	82.85
		7	115	M3	82.64
		6	98	M3	81.79
		5	82	M3	80.05
		4	66	M3	78.60
		3	49	M3	70.24
		2	33	M3	44.14
KWS	FP	32	n/n	None	86.75
	FX	8	304	M7	86.38
		7	266	M7	85.68
		6	228	M4	85.41
		5	190	M4	81.90
		4	152	M4	77.19
		3	114	M3	61.70
		2	76	M3	9.07
FER	FP	32	n/n	None	66.48
	FX	8	654	None	65.34
		7	572	None	64.59
		6	491	M7	65.03
		5	409	M7	62.41
		4	327	M7	57.65
		3	246	M4	18.22
		2	164	M4	17.44

scheme consists of the same procedure deployed in the **VQ** flow during the H-bit Quantization stage. For each benchmark, the first row collects the classification accuracy of the original 32-bit floating-point model (FP). The next seven rows quantify the figures of merit of the n-bit quantization with $n \in [2-8]$. The column *R*AM shows the n-th model memory footprint computed with the memory model introduced in Section III-B (inline with results in [17]). As an additional piece of information, the *C*ore column reports the smallest ARM Cortex-M with enough RAM to host and run the CNN (column Core). The *T*op-1 column collects the top-1 accuracy achieved by each model.

As demonstrated in previous works (e.g. [12]), 8-bit quantization reaches almost the same accuracy of the original FP model. This is also confirmed by our experiments (the worst-case loss is 1.14% for FER). Lower bit-widths show a substantial quality degradation: e.g. with 3-bit the accuracy loss ranges from 12.56% (for IC) to 48.26% (for FER). The results confirm that quantization is an effective compression strategy. For instance, for FER, the 6-bit model would fit in cores with 512 KB RAM (e.g. the M7) still guaranteeing a reasonable accuracy loss (1.45%). This analysis does not consider the lack of proper hardware architectures to process workloads with less than 8-bit. As already discussed, there are alternative software patches, but they proved to be very impractical [14].

Fig. 3 highlights the key results obtained with the proposed **VQ** as it gives a fair comparison w.r.t. n-bit quantization **Q**, our baseline. For the sake of completeness, **VQ** is also compared against a classical pruning methodology (label **P**) whose details are introduced later. The bars show the top-1 accuracy loss, which is defined as the difference between the top-1 accuracy of the original FP model (first row of each benchmark in Table III) and that achieved after the compression. Each bit-width is associated with a different memory footprint, those obtained with the theoretic fixed-point n-bit quantization (i.e.

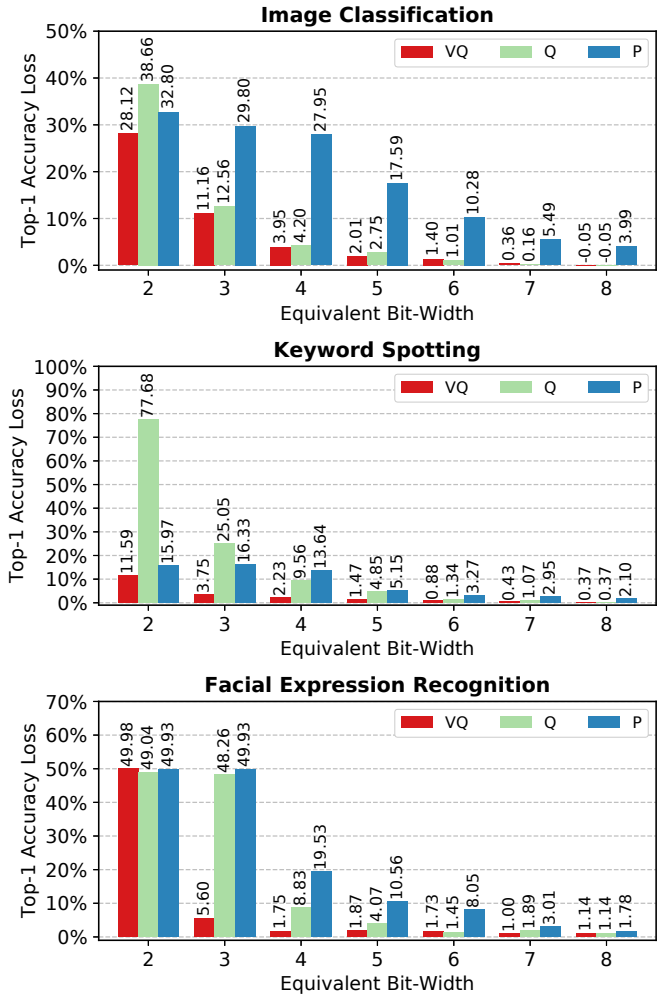

Figure 3: Top-1 accuracy loss in the three different applications (lower is better). The baseline accuracies are the same reported in Table III.

those in Table III). For the three applications and all the equivalent bit-widths, **VQ** converges to solutions that meet the same memory footprint of **Q**, yet achieving much lower losses (e.g. 7.08% less for FER at 4-bit). In other words, **VQ** makes pure theoretic n-ary quantization a practical solution even on general purpose MCUs. Some exceptions might exist, like FER with 6-bit, where the **VQ** loss (1.73%) is larger than that of Q (1.45%). However, the gap is small (0.28%). The greedy nature of the heuristic and the statistic retraining stages are the first sources of such mismatch. Even more interesting, there are cases where **VQ** outperforms **Q** using fewer bits. For instance, KWS compressed with **VQ** set to 4-bits shows a loss (2.23%) that is smaller than that of **Q** with 5-bit (4.85%). The same holds for FER, which is the benchmark with the highest complexity and the largest memory footprint: **VQ** with 3 bits (5.60%) vs. **Q** with 4-bits (8.83%). That's due to the quantization-aware compression method integrated into **VQ**: it is aware of the underlying hardware constraints, hence it runs a well-balanced filter reduction.

One may argue that "memory-equivalence" can be achieved with any standard compression method (e.g. any standard

accuracy-driven pruning) enhanced with the ability to meet a specific memory constraint. For such reason, Fig. 3 also reports the result obtained with a "blind" pruning (label **P**) where filters are dropped one-by-one ($\ell 1$-norm as a ranking criterion) till the CNN fits the same RAM of that obtained with Q. Differently from **VQ**, the pruning strategy does not involve any quantization awareness and filters are kept to maximum precision (16 bits for this case of study). The results clearly show that P meets the memory constraint of any equivalent bit-width but it gets worse in terms of accuracy. Moreover, its efficacy reduces with tight memory constraints. The worst-case is for FER with 3-bit, where the accuracy loss is 44.33% larger than that of **VQ**. The main reason is that pruning alone removes the filters at a too fast pace to reach the target memory footprint; this has dramatic effects on the classification accuracy.

As a final comment, we noticed that models which undergo the **VQ** flow gets faster with the reduction of the equivalent bit-width. This is a direct effect of the compression method which, unlike quantization, reduces the number of memory accesses (both read and write) and the number of operations without requiring any extra code or special routine. Considering KWS for instance, with an accuracy loss of 1% (equivalent to 6-bit Q) the CNN model takes 25% less memory and is 1.4x faster.

V. Conclusion

Virtual quantization (**VQ**) is a compression method for embedded CNNs. It has been built with the specific goal of emulating arbitrary bit-width arithmetic on those memory-constrained architectures that would benefit most from quantization but cannot implement it due to the lack of dedicated hardware units. Experimental results conducted on three real-life applications demonstrated the feasibility of the proposed method. If compared to CNNs squeezed with state-of-the-art n-bit quantization, those that undergo the **VQ** flow meet the same memory budget while achieving much higher quality-of-results.

VI. Acknowledgment

Co-funded by Compagnia di San Paolo

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[2] [Online]. Available: https://os.mbed.com/platforms

[3] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

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

[5] Challenges in representation learning: Facial expression recognition challenge. [Online]. Available: http://www.kaggle.com/

[6] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," in *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*. IEEE, 2016, pp. 1–8.

[7] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," in *neural networks (IJCNN), 2017 international joint conference on*. IEEE, 2017, pp. 2547–2554.

[8] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[9] V. Peluso and A. Calimera, "Scalable-effort convnets for multilevel classification," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[10] M. Grimaldi, V. Tenace, and A. Calimera, "Layer-wise compressive training for convolutional neural networks," *Future Internet*, vol. 11, no. 1, p. 7, 2019.

[11] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[12] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[13] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, no. 3, pp. 339–354, 2016.

[14] M. Rusci, A. Capotondi, F. Conti, and L. Benini, "Quantized nns as the definitive solution for inference on low-power arm mcus?: work-in-progress," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 2018, p. 12.

[15] Y. Umuroglu, L. Rasnayake, and M. Själander, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 307–3077.

[16] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[17] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[18] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "Lognet: Energy-efficient neural networks using logarithmic computation," in *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 2017, pp. 5900–5904.

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

[20] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," *Energy*, vol. 41, p. 46, 2018.

[21] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *arXiv preprint arXiv:1705.08922*, 2017.

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

[23] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[24] Gnu arm embedded toolchain. [Online]. Available: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

[25] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[28] T. N. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.