Moving Convolutional Neural Networks to Embedded Systems: the AlexNet and VGG-16 case

Cesare Alippi[*]
Politecnico di Milano
Milano, Italy
cesare.alippi@polimi.it

Simone Disabato
Politecnico di Milano
Milano, Italy
simone.disabato@polimi.it

Manuel Roveri
Politecnico di Milano
Milano, Italy
manuel.roveri@polimi.it

ABSTRACT

Execution of deep learning solutions is mostly restricted to high performing computing platforms, e.g., those endowed with GPUs or FPGAs, due to the high demand on computation and memory such solutions require. Despite the fact that dedicated hardware is nowadays subject of research and effective solutions exist, we envision a future where deep learning solutions -here Convolutional Neural Networks (CNNs)- are mostly executed by low-cost off-the shelf embedded platforms already available in the market.

This paper moves in this direction and aims at filling the gap between CNNs and embedded systems by introducing a methodology for the design and porting of CNNs to limited in resources embedded systems. In order to achieve this goal we employ approximate computing techniques to reduce the computational load and memory occupation of the deep learning architecture by compromising accuracy with memory and computation.

The proposed methodology has been validated on two well-know CNNs, i.e., AlexNet and VGG-16, applied to an image-recognition application and ported to two relevant off-the-shelf embedded platforms.

KEYWORDS

Embedded Systems, Deep Learning, Convolutional Neural Networks, Approximate Computing.

ACM Reference Format:
Cesare Alippi, Simone Disabato, and Manuel Roveri. 2018. Moving Convolutional Neural Networks to Embedded Systems: the AlexNet and VGG-16 case. In *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN) (IPSN'2018).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[*]Cesare Alippi is also with Universitá della Svizzera Italiana (USI), Switzerland.

1 INTRODUCTION

The ongoing technological (re-)evolution has made *embedded systems* pervasive to constitute the main building blocks for cutting-edge distributed application scenarios (e.g., Internet-of-Things, Industry 4.0, Smart Homes and Cities, Intelligent transportation). However, in order to design effective, efficient and credible real-world solutions (say out-of-lab solutions), pervasive computing systems must be endowed with different levels of intelligence in the data acquisition and processing phases as well as in the decision-making one [29][1]. In fact, moving intelligence to the datastream-production level (i.e., directly at the sensor or cluster of sensors level) allows the pervasive computing system to reduce the decision-making latency and the communication bandwidth and improve the energy-efficiency [4][6].

For these reasons, the interest associated with the embedding of *intelligence in embedded systems* is steadily increasing over time, leading to a wide range of machine learning and computational intelligence-based solutions for embedded platforms (e.g., adaptive sampling algorithms [2], model-free fault detection and diagnosis mechanisms [5]). The interest here is in providing intelligent solutions executable as extra functions in embedded systems and, as such, designed to take into account constraints on memory, computation and energy. Different levels of intelligence are also part of the application, for instance, as it happens in deep learning/convolutional neural networks-based solutions that represent the state-of-the-art in many recognition/classification applications. Differently from traditional "shallow" solutions, where features are designed manually, deep learning techniques learn both the needed features and the classification task [26]. Among the wide range of deep learning solutions, this paper focuses on *Convolutional Neural Networks* (CNNs) that, among the others, represent the state-of-the-art solutions for image recognition (as well as video classification, natural language processing, drug discovery and modelling artificial players in games [8]). AlexNet [31], VGG-16/VGG-19 [40], OverFeat [38], GoogLeNet [43], ResNet [25] are some of the most popular CNNs present in the literature (further CNNs can be found in [45], [32] and [47]). The deep architecture of these CNNs is generally organized into a pipeline of descriptors characterized by an increasing granularity of the representation, i.e., from edges and corners to motifs, parts and, finally, to objects. To achieve this goal, CNNs rely on multiple convolutional and sub-sampling layers to extract the features and reduce their dimensionality, respectively, and on classification modules (e.g., fully connected and softmax) to provide the final classification of the input image. For these reasons, CNNs are typically characterized by a high computational load and memory occupation. For example, the deep architecture of AlexNet encompasses approximately 61 million parameters requiring more than 230 MB for the storage of the network weights only and more than 725 million floating-point operations to provide an output; VGG-16 is even more demanding in terms of memory and computation (i.e., more than 138 million parameters requiring more than 527 MB for the storage and more than 13 billion operations to process the input image).

It is obvious that the requested high computational load and memory occupation find in Graphics Processing Units (GPUs) or custom hardware (Field-Programmable Gate Array -FPGA- or ad-hoc hardware)[17] a natural hosting platform. On the other side, embedded systems are generally characterized by technological constraints on memory and computation (i.e., the amount of RAM memory ranges from few KB to few MB and the clock frequency from dozens to hundreds of MHz). For these reasons, to the best of our knowledge, embedded systems have never been considered as viable technological solutions for CNNs and the use of CNNs as well as other deep learning techniques in such systems still represents a completely unexplored research field [17] despite of its utmost relevance.

The aim of this paper is to fill the gap between CNNs and embedded systems by introducing a methodology to design application-specific and approximated CNNs able to be executed in embedded systems characterized by strong constraints on memory, computation and energy consumption. The contributions of the paper are:

- a methodology to design application-specific and approximated CCNs able to be executed in off-the-shelf embedded systems;
- a novel filter-selection mechanism for approximated CNNs able to activate only the convolutional filters useful or requested for the application-specific classification problem;
- the first porting of an application-specific and approximated CNN to embedded platforms, here an STM32-F7 and a Raspberry Pi 3B.

The paper is organized as follows. Section 2 describes the related literature. The approximated versions of CNNs are introduced in Section 3, while the proposed methodology for designing approximated CNNs is detailed in Section 4. The experimental results and the desing/development of an image-recognition embedded application on the STM32-F7 and Raspberry Pi 3B embedded platforms are described in Section 5. Comments and future works are finally drawn in Section 6.

2 RELATED LITERATURE

The literature is plenty of solutions aiming at reducing the computation and memory requirements of CNNs by means of approximation techniques. Such solutions typically operate during the training phase of CNNs and allow to reduce the training time that is typically very high for deep architectures (i.e., from hours to weeks on high-performance computing systems) [14]. For example, [33][34] introduce approximations in the sequence of convolutional and max-pooling layers by proposing joint convolutional/sub-sampling layers. Differently, separable convolutional filters have been introduced in the training phase [37] to learn sparse representations of the inputs. Unfortunately, reducing the complexity of the training phase does not impact the operational phase since CNNs are very expensive from the computational point of view even during the processing of the input images [17].

A relatively large literature about precision scaling in CNNs exists [42]. For example, the idea of compressing the parameters of the convolutional layers and fine-tuning the fully-connected layers has been proposed in [14] and [7], while [23] focuses on reducing the memory occupation of CNNs through pruning, weight quantization and coding techniques. These solutions provide meaningful reductions in terms of memory occupation but, unfortunately, the reduction in computational load (provided, for example, by the use of specific libraries for sparse representation or by smaller memory footprints reducing the data transfer overhead) is very limited.

Conversely, the literature about CNNs for embedded systems is very limited and available solutions refer to either general-purpose or custom embedded hardware [17].

General-purpose hardware is becoming very popular thanks to its reduced costs, flexibility and easy-to-use properties. [13] and [39] introduce optimized versions of CNNs for general-purpose CPUs implementing cross-image parallelization via software through multi-threading. Differently, [11] and [44] propose modifications to CNNs so as to increase the speed of processing images via unrolling convolutional operations and the quantization of the network weights. Despite the advantages provided by these solutions, their use in general-purpose embedded systems is still unfeasible due to the high computational and power-consumption demand.

Interestingly, in the field of general-purpose hardware, GPUs gained importance in the recent years providing high-speed processing and parallelism [30] [12]. Unfortunately, GPUs are typically not present in embedded systems.

Differently, *custom hardware* relies on ad-hoc designed platforms. For this reason, it typically provides better performance and less power consumption than general-purpose hardware at the expense of larger costs in the design phase. While designing hardware solutions for CNNs from the scratch (e.g., see [10]) could be a very complex operation requiring high skills and expertise, FPGA solutions have been recently proposed to support the design of ad-hoc hardware for CNNs. In this research stream, [46] introduced a FPGA-based accelerator for CNNs implementing floating-point operations for the convolutional layers. A FPGA-based streaming method for CNNs has been recently proposed in [17]. This work introduces also a compiler to transform high-level representations of CNNs into executable codes for FPGA able to support data reuse and concatenation for hardware acceleration. Other FPGA-based solutions can be found in [28], [19] and [16]. Unfortunately, the presence of FPGAs in embedded systems is still very limited.

As a final remark, in spite of the wide range of algorithmic and technological solutions available to reduce the computational load and memory occupation of CNNs, their use in general-purpose embedded systems is still not a viable technological solution. This paper provides a first relevant contribution in this direction.

3 CONVOLUTIONAL NEURAL NETWORKS FOR EMBEDDED SYSTEMS

In order to design and develop CNNs able to operate within the strict requirements of embedded systems (in terms of memory occupation and computational load), this paper introduces *application-specific and approximated CNNs able to run on embedded systems.* More specifically, the proposed solutions represent transformed versions of state-of-the-art CNNs that have been:

- tailored to an application-specific classification problem, i.e., proposed solutions are able to solve a large class of applications by relying on the general-purpose features extracted by CNNs and a classifier trained on the specific application at hand. To achieve this goal, the classification layer of the original CNNs is replaced by a classifier trained on the given image-classification problem;
- approximated in terms of architecture and weight precision w.r.t. the original CNNs so as to fit in a given embedded-system platform. To achieve this goal, the original CNNs is pruned, while the wordlenght for the weights is reduced.

The idea of considering available CNNs relies on the fact that these networks represent the state-of-the-art in many image-recognition scenarios. More specifically, we want to exploit the intrinsic ability of CNNs to automatically extract features with increasing complexity and meaning. In fact, as mentioned in Section 1, CNNs extract low-level features such as edges and corners in the first layers that are hierarchically organized, in next layers, into motifs, parts and, finally, into high-level "objects". Despite the fact that they have been configured on the dataset used to train the CNN, this hierarchy of features could be general enough to be used even in other image-classification scenarios [36]. On the one hand, this might lead to sub-optimal solutions w.r.t. CNNs trained from scratch for a specific classification problem. On the other hand, the re-training of CNNs would be infeasible in real-world scenarios of embedded systems since the computational power is limited. Moreover, relying on ready-to-use CNNs as "general-purpose" feature extractors would allow us to consider the designed CNNs for embedded systems even in time-varying scenarios or environment [15], i.e., those affected by *concept-drift* inducing changes in the probability density functions of the classes. In this way we can adapt the application-specific classification step of the designed approximated CNNs in responses to changes in the data-generating process (i.e., following an active, passive or hybrid adaptive approach [15]) without requiring the re-training of the convolutional layers.

More formally, let $\mathcal{I}$ be a $m \times n \times c$ input image of m rows, n columns and c channels (i.e., $c = 1$ for gray-scale images and $c = 3$ for RGB images). Let $y \in \Lambda = \{\omega_1, \ldots, \omega_\Lambda\}$ be the output of the CNN representing the multi-class classification of $\mathcal{I}$. The given CNN Φ is defined as

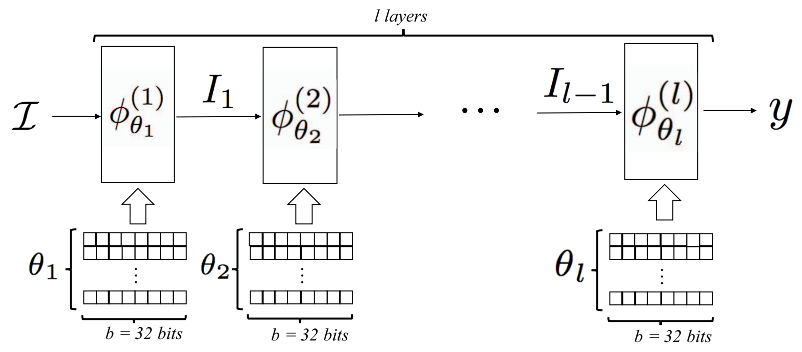
$$y = \Phi(\mathcal{I}) \tag{1}$$

where

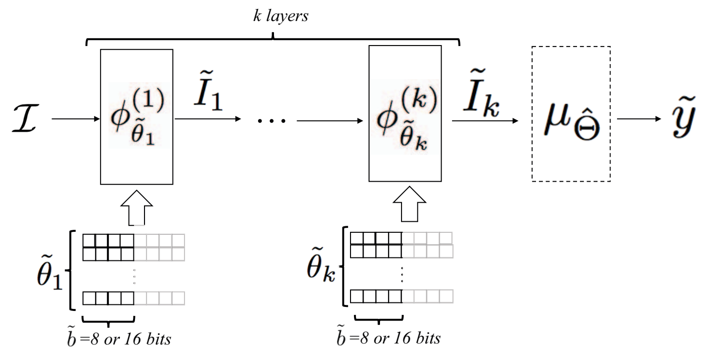
$$\begin{aligned} \Phi(\mathcal{I}) &= \phi^{(l)}_{\theta_l}(I_{l-1}) \\ I_j &= \phi^{(j)}_{\theta_j}(I_{j-1}),\ \text{with } j = 2, \ldots, l-1 \\ I_1 &= \phi^{(1)}_{\theta_1}(\mathcal{I}), \end{aligned}$$

l is the number of layers, $\phi^{(i)}_{\theta_i}$, $i = 1, \ldots, l$, is the function parametrized in θ_i accounting for i-th layer of the CNN and I_i the output of the i-th layer to be processed by the next $i+1$-th layer. $\phi^{(i)}_{\theta_i}$s can be convolutional filters, Max-pooling operators, Rectified Linear Unit (ReLU), Fully-Connected layers or Softmax layers. Details about the AlexNet and the VGG-16 are given in Table 1.

Given function Φ we want to identify the application-specific and approximated version $\widetilde{\Phi}$ of Φ. To achieve this goal, as shown in Figure 1, Φ undergoes the following two approximation mechanisms [35]:



(a) The general architecture of a Convolutional Neural Network.



(b) The general architecture of a Convolutional Neural Network for Embedded Systems.

Figure 1: The architectures of Convolutional Neural Networks and Convolutional Neural Networks for Embedded Systems

3.0.1 *Task dropping.* As described in [35], the goal of task dropping is to reduce the computational load and the memory occupation by skipping the execution of some tasks associated with the processing chain. Here, instead of considering the whole chain of l layers of Φ, we keep in $\widetilde{\Phi}$ only the first k layers[1]. As shown in Figure 1(b), a classification algorithm μ_Θ parametrized in the vector $\Theta \in \mathbb{R}^\Theta$ is appended at the end of the k-th layer to accomplish the classification of $\mathcal{I}$ based on the activations stored in $\tilde{I}_k$. The classifier can be a Support Vector Machine, a Decision Tree, a Feed-forward Neural Network, etc.... In the approximated version, μ_Θ takes the role of the fully-connected and softmax layers that typically perform the classification phase in Φ. To achieve this goal, μ_Θ is configured on a training set $T = \{(\mathcal{I}_1, y^o_1), \dots, (\mathcal{I}_N, y^o_N)\}$ that is provided to $\widetilde{\Phi}$, where $\mathcal{I}_i$ is the i-th input image and $y^o_i \in \tilde{\Lambda} = \{\tilde{\omega}_1, \dots, \tilde{\omega}_{\tilde{\Lambda}}\}$ the associated class. More specifically, μ_Θ is configured on $\{(\tilde{I}^1_k, y^o_1), \dots, (\tilde{I}^1_N, y^o_N)\}$, where $\tilde{I}^i_k$ is the value of the activations at the output of the k-th layer when considering the input image $\mathcal{I}_i$. We emphasize that the classification problem to be addressed by $\widetilde{\Phi}$ is generally different from the one of Φ, hence the set of classes $\tilde{\Lambda}$ to be learned by μ_Θ is not the one used to train Φ (e.g., the AlexNet is trained on the 1000 classes of the ILSCVRC dataset, while μ_Θ could be trained on two or more classes not belonging to ILSCVRC).

[1]Different strategies of task dropping can be envisioned, e.g., by selecting non-consecutive layers of Φ.

3.0.2 *Precision scaling.* The aim of precision scaling is to change the precision (number of bits for the representation) of the inputs or intermediate operands to reduce the memory occupation [35]. In our specific case, precision scaling aims at reducing the memory occupation associated with the weights of Φ by considering approximated versions $\widetilde{\theta_i}$ of θ_i. In fact, as stated in [21], weights θ_is are typically real numbers and standard implementations in Φ encompass 32-bit data types (floating point). Precision scaling, which is computed through the rounding of weights θ_is, aims at employing a fixed-point representations of the weights so as to consider 16-bit or 8-bit data types (e.g., *int* or *short int*). This aspect will be clarified in the next section where the memory occupation of $\widetilde{\Phi}$ is detailed. Interestingly, there is a parallelism between precision scaling and CNN regularization. On one hand, the rounding operation of precision scaling introduces a perturbation in the original weights θ_is [1]. On the other hand, noise-addition or dropout mechanisms [41] are typically considered during the training of CNNs to make them more robust against perturbations, e.g., more tolerant to truncation/rounding operations. Other techniques to achieve similar level of robustness could encompass the modification of the figures of merit to be optimized during learning [41].

Given the above notation, the procedure $\mathcal{T}$ transforms Φ into $\widetilde{\Phi}$

$$\widetilde{\Phi} = \mathcal{T}\left(\Phi, k, q, T, \mu_\Theta\right) \tag{2}$$

where Φ is the original CNN, k the number of considered layers, q the number of decimal digits to be rounded, T the training set and μ_Θ the classification algorithm to be considered. The output $\widetilde{\Phi}$ of $\mathcal{T}$ is the approximated CNN defined as follows

$$\begin{aligned} \tilde{y} =& \widetilde{\Phi}(\mathcal{I}) = \mu_{\hat{\Theta}}(\tilde{I}_k) \\ \tilde{I}_j &= \phi^{(j)}_{\tilde{\theta}_j}(\tilde{I}_{j-1}), \text{ with } j = 2, \dots, k \\ \tilde{I}_1 &= \phi^{(1)}_{\tilde{\theta}_1}(\mathcal{I}) \end{aligned} \tag{3}$$

where $\tilde{y} \in \tilde{\Lambda}$, $\tilde{I}_i$ is the output of the i-layer of $\widetilde{\Phi}$, and the parameter vector $\hat{\Theta}$ of μ_Θ has been trained on

$$\{(\tilde{I}^1_k, y^o_1), \dots, (\tilde{I}^N_k, y^o_N)\}$$

being $\tilde{I}^i_k$ the output of the k-layer associated with the i-th image of T.

4 DESIGNING APPROXIMATED CONVOLUTIONAL NEURAL NETWORKS

The proposed methodology aims at designing application-specific and approximated $\widetilde{\Phi}$ of Φ as defined in Eq. (3). Section 4.1 characterizes the computational load and memory occupation of Φ and $\widetilde{\Phi}$, while Section 4.2 describes the proposed methodology. A further level of approximation for $\widetilde{\Phi}$ is introduced in Section 4.3 to consider, in the k-th layer, only the convolutional filters useful for the application-specific classification problem; this allows to further reduce computational load and memory occupation of $\widetilde{\Phi}$.

4.1 Computational complexity and memory occupation

Given Φ and $\widetilde{\Phi}$ defined in Eq. (1) and Eq. (3), respectively, we can now evaluate their computational load and memory occupation. More specifically, the memory occupation of Φ and $\widetilde{\Phi}$ can be defined as

Table 1: Characteristics of AlexNet and VGG-16. l is the number of layers, N^Φ is the number of weights to be stored and b the number of bits considered for each weight. M^Φ and C^Φ are the memory occupation and the computational load, respectively.

CNN	AlexNet	VGG-16
l	23	39
N^Φ	60.9×10^6	138.3×10^6
b	32	32
M^Φ	232.5 MB	527.8 MB
C^Φ	725.2×10^6	13626.2×10^6

$$M^\Phi = N^\Phi b \tag{4}$$

and

$$M^{\widetilde{\Phi}} = N^{\widetilde{\Phi}} \tilde{b}, \tag{5}$$

where N^Φ and $N^{\widetilde{\Phi}}$ are the number of weights to be stored in Φ and $\widetilde{\Phi}$, respectively, and b and $\tilde{b}$ are the memory wordlengths in terms of number of bits required to store each weight in Φ and $\widetilde{\Phi}$, respectively. The overhead required to store the data structures in typically negligible compared to the one requested by weights. Typically, in Φ, $b = 32$ bit since the single-precision floating point-data type is used [21]. Without any loss of generality we assume that all the weights in Φ have the same wordlength. The precision scaling mechanism allows to achieve $\tilde{b} < b$ when the rounded fixed-point representation of $\tilde{\theta}_i$ can be stored with $\tilde{b} = 8$ or $\tilde{b} = 16$ bits [1], i.e., when $\tilde{\theta}^j_i \cdot 10^{p-q}$ with $j = 1, \dots, k$ and $i = 1, \dots, |\tilde{\theta}^j|$ belong to the interval $[-2^{\tilde{b}-1}, 2^{\tilde{b}-1} - 1]$. The memory reduction when considering $\widetilde{\Phi}$ instead of Φ depends explicitly on k and implicitly on q and can be quantified as

$$\Delta M = M^\Phi - M^{\widetilde{\Phi}} = N^{\widetilde{\Phi}}(b - \tilde{b}) + (N^\Phi - N^{\widetilde{\Phi}})b. \tag{6}$$

As suggested in [24], the computational load of Φ can be approximated as the sum of the computational load of the convolutional filter C^Φ_{conv} and the one of the fully connected C^Φ_{fc} layers, i.e.,

$$C^\Phi = C^\Phi_{conv} + C^\Phi_{fc}, \tag{7}$$

while the computational loads introduced by pooling and ReLU layers can be typically neglected [18]. Without any loss of generality the computational load is measured as the number of multiplications required to accomplish the layers goals[2]. The convolution computational load C^Φ_{conv} of Φ is defined as

$$C^\Phi_{conv} = \sum_{i=1}^{N_c} n_{i-1} \cdot s^2_i \cdot n_i \cdot m^2_i, \tag{8}$$

[2]Other figures of merit could be considered as well.

where N_c is the number of convolutional layers in Φ, n_i is the width of the i-th layer, i.e., the number of filters, n_{i-1} represents the number of input channels at the convolutional layer i, and s_i and m_i refer to the spatial size of the filter and the spatial size of the output feature map of the i-th layer, respectively. The computational load C_{fc}^{Φ} of the fully-connected layers is the product between the number of neurons n_i of the layer i and the one n_{i-1} of the layer $i-1$, i.e.,

$$C_{fc} = \sum_{i=1}^{N_{fc}} n_{i-1} \cdot n_i \tag{9}$$

where N_{fc} is the number of fully connected layers in Φ.

Similarly to Eq. (7), the computational load of $\widetilde{\Phi}$ can be defined as

$$C^{\widetilde{\Phi}} = \tilde{C}_{conv} + \tilde{C}_{\mu}, \tag{10}$$

where

$$\tilde{C}_{conv} = \sum_{i=1}^{\widetilde{N}_c} n_{i-1} \cdot s_i^2 \cdot n_i \cdot m_i^2 \tag{11}$$

being $\widetilde{N}_c \leq k$ the number of convolutional layers in $\widetilde{\Phi}$ and $\tilde{C}_{\mu}$ is the computational load of μ_{Θ}. In most of the cases (e.g., by considering SVMs, Decision Trees, FFNNs) [9], $\tilde{C}_{conv} \gg \tilde{C}_{\mu}$, hence

$$C^{\widetilde{\Phi}} \simeq \tilde{C}_{conv}. \tag{12}$$

The reduction in computational load when considering $\widetilde{\Phi}$ depends only on k (being $\widetilde{N}_c$ the number of convolutional layers among the k layers of the $\widetilde{\Phi}$) and can be computed as follows

$$\Delta C = C^{\Phi} - C^{\widetilde{\Phi}} = \sum_{i=\widetilde{N}_c+1}^{N_c} n_{i-1} \cdot s_i^2 \cdot n_i \cdot m_i^2 + C_{fc}^{\Phi}. \tag{13}$$

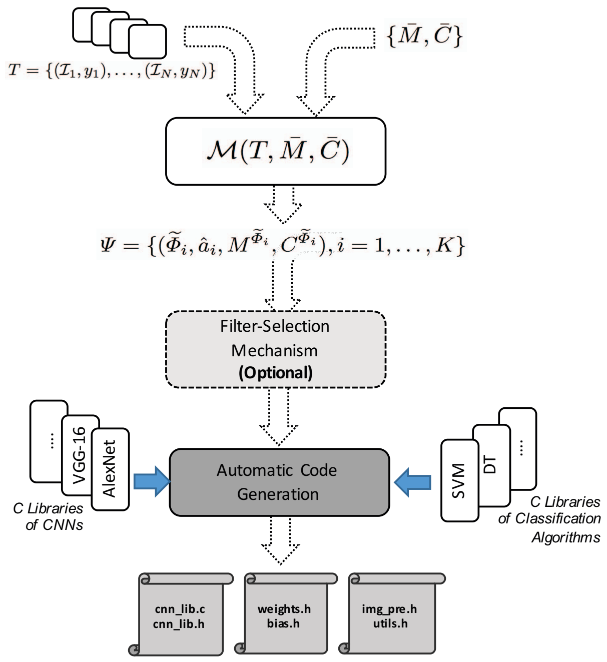
As expected, task dropping provides a reduction in both memory occupation and computational load, while precision scaling allows for reducing only memory occupation.

Details about the memory occupation and computational load of AlexNet and VGG-16 are given in Table 1. In Table 2 details about the corresponding $\widetilde{\Phi}$ for an image-recognition application targeted to two off-the-shelf embedded platforms are given.

4.2 The proposed methodology

The proposed methodology $\mathcal{M}$, shown in Figure 2, aims at designing $\widetilde{\Phi}$ able to run on a given embedded-system platform. To achieve this goal, $\mathcal{M}$ receives in input a dataset T and the constraints on memory $\overline{M}$ and computation $\overline{C}$ provided by the embedded-system application designer. The output Ψ of $\mathcal{M}$ is the Pareto set of $\widetilde{\Phi}$ satisfying the constrains on $\overline{M}$ and $\overline{C}$. More formally, $\mathcal{M}$ is defined as

$$\Psi = \mathcal{M}(T, \bar{M}, \bar{C}) \tag{14}$$



$T = \{(\mathcal{I}_1, y_1), \ldots, (\mathcal{I}_N, y_N)\}$
$\{\bar{M}, \bar{C}\}$
$\mathcal{M}(T, \bar{M}, \bar{C})$
$\Psi = \{(\widetilde{\Phi}_i, \hat{a}_i, M^{\widetilde{\Phi}_i}, C^{\widetilde{\Phi}_i}), i = 1, \ldots, K\}$
Filter-Selection Mechanism (Optional)
...
VGG-16
AlexNet
C Libraries of CNNs
Automatic Code Generation
SVM
DT
...
C Libraries of Classification Algorithms
cnn_lib.c cnn_lib.h
weights.h bias.h
img_pre.h utils.h

Figure 2: The proposed methodology for designing convolutional neural networks for embedded systems.

where

$$\Psi = \{(\widetilde{\Phi}_i, \hat{a}_i, M^{\widetilde{\Phi}_i}, C^{\widetilde{\Phi}_i}), i = 1, \ldots, N^{\Psi}\}$$

stores the N^{Ψ} $\widetilde{\Phi}$ representing the Pareto frontier w.r.t. the estimated accuracy $\hat{a}_i$ on T, the memory occupation $M^{\widetilde{\Phi}_i}$ and the computational load $C^{\widetilde{\Phi}_i}$ defined in Eq. (5) and (12), respectively.

In the process of identifying Ψ, $\mathcal{M}$ operates in two different steps.

Step 1. Starting from a given Φ, we define as Υ_I the set of $\widetilde{\Phi}$ satisfying the constraints on $\overline{M}$ and $\overline{C}$, i.e., ,

$$M^{\widetilde{\Phi}^{k,p,\mu_\Theta}} \leq \overline{M}$$
$$C^{\widetilde{\Phi}^{k,p,\mu_\Theta}} \leq \overline{C}$$

obtained through $\mathcal{T}$ and the exploration of $k = \{1, \ldots, l\}$, $q = \{0, \ldots, q_{max}\}$, and $\mu_\Theta \in \{\mu_\Theta^1, \ldots, \mu_\Theta^{max}\}$. q_{max} is the user-defined maximum number of fractional digits to be rounded in the precision scaling phase[3]. Obviously, once a fixed-point data-type representation is considered, q_{max} is related to the maximum number of bits that are used to store the fractional part of weights $\tilde{\theta}_i$s. The set of classification

[3]In the experimental analysis in Section 5, $q_{max} = 5$. The selection of q could be further optimized by considering different values of q for different layers or filters of $\widetilde{\Phi}$.

algorithms to be considered is $\{\mu_\Theta^1, \ldots, \mu_\Theta^{max}\}$. In Eq. (14), we are considering a single Φ but the whole methodology encompasses multiple Φs in input (as shown in Section 5).

Step 2. We estimate the accuracy $\hat{a}_i$ of each $\widetilde{\Phi}_i$ in Υ_I and provide in Ψ the Pareto frontier w.r.t. $\hat{a}$, $M^{\widetilde{\Phi}}$ and $C^{\widetilde{\Phi}}$. In this step we estimate the parameter vector $\hat{\Theta}$ by means of a training procedure on T. The computation of $\hat{a}_i$ is performed through sample-partitioning but different techniques could be considered as well (e.g., Leave-One-Out, K-fold Cross-validation, or Bootstrap). As pointed out in [3], the comparison between the estimated accuracies of two $\widetilde{\Phi}$ should in principle take into account also the confidence intervals of the estimate (especially when the number of samples is small). This can be easily managed within $\mathcal{M}$ by considering a paired test to assess the performance comparison instead of simply comparing the estimated accuracies [3].

Once a specific $\widetilde{\Phi}$ has been selected from Ψ by the embedded-system designer, it can be automatically translated into C-language source code and libraries (both .c and .h are provided). This step can be easily made automatic since, given $\widetilde{\Phi}^{k,p,\mu_\Theta}$, all the layers and weights and the classification algorithm have been functionally defined. Hence, they can be easily imported by pre-defined C libraries storing ready-to-use CNNs (e.g., VGG-16 and AlexNet, etc..) or classification algorithms (e.g., SVM, Decision Tree, etc...). The C-language source code and libraries representing the chosen $\widetilde{\Phi}$ can be easily inserted into software projects for designing the embedded systems endowed with intelligent abilities[4].

4.3 Filter-selection mechanism

Once a specific $\widetilde{\Phi}$ in Ψ has been selected by the embedded-system designer, we can perform an additional approximation to further reduce computational load and memory occupation on the given embedded-system platform. More specifically, when the k-th layer of $\widetilde{\Phi}^{k,p,\mu_\Theta}$ is a convolutional layer (but this can be easily extended to all the layers whose processing does not require inter-filtering operations, e.g., cross-filter normalization, since the last convolutional layer), we can apply a filter-selection procedure to identify those filters providing output features that are truly beneficial to support the classification by μ_Θ and discard the others (i.e., those with negligible or negative effects on the specific classification problem).

This task is very close to the feature selection techniques present in machine learning [22] but, here, the goal is to minimize the number of filters to be activated (and not just the number of features to be computed). To achieve this goal we ranked all the features in $\tilde{I}_k$ according to the classification accuracy computed on T by considering one feature at a time as input of μ_Θ. Then, we set the maximum number F^{max} of filters to be activated[5] and, for each $f = 1, \ldots, F^{max}$, we start selecting the first feature of the ranking as well as all the other features belonging to the same filter and we remove them from the ranking. We repeat this step up to when f filters have been selected. Finally, we train μ_Θ with the features coming from the f filters previously selected. The value of f providing the highest classification accuracy is finally selected.

This filter-selection optimization could be beneficial for $\widetilde{\Phi}$ from three main perspectives. *First*, we can avoid the storage of the weights of those filters of the k-th layer that have not been selected, hence reducing the memory occupation of $\widetilde{\Phi}$. *Second*, reducing the amount of filters to be executed allows us to significantly reduce the computational load of the k-th layer and, thus, of the whole $\widetilde{\Phi}$. *Third*, it is possible to train μ_Θ on a smaller set of features (i.e., those providing the highest class-specific information), hence reducing the curse-of-dimensionality problem [22] and, generally, providing higher classification accuracies.

5 EXPERIMENTAL RESULTS

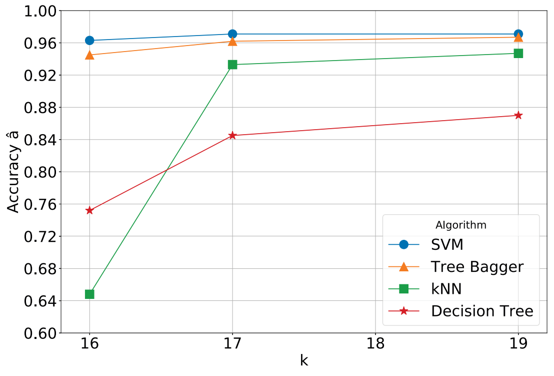
We evaluated the effectiveness of the proposed methodology in two main steps. At first, we tested the proposed methodology on a wide synthetic analysis encompassing different configurations of the number of layers k, the number of decimal digits to be rounded q and the classification algorithms μ_Θ. In this step the considered CNNs are the AlexNet and the VGG-16[6] . The results of this analysis are described and commented in Section 5.1. Afterwards, in Section 5.2, we applied the proposed methodology to port the AlexNet to two off-the-shelf embedded platforms.

The dataset used for all the experiments is the *Caltech-256* benchmark [20] containing more than 256 image classes and at least 80 images per class. It is worth noting that this benchmark is not the one used to train the AlexNet or the VGG-16 that have been trained on ILSVRC [27]. The set of machine learning algorithms considered for μ_Θ is: SVM, Tree Bagger (with 100 Decision Trees), k-Nearest Neighbour (where k is set as the square root of the number of samples in the training set), and Decision Tree.

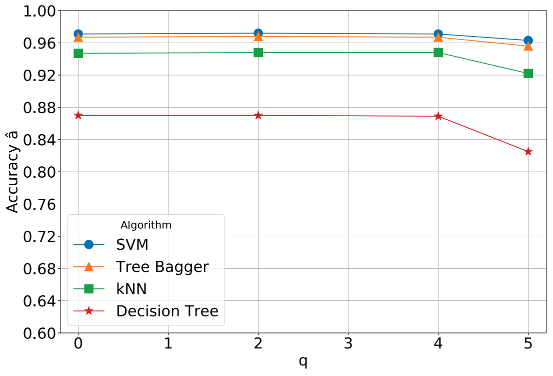
[4]The code of the proposed methodology is made available to the scientific community as a Matlab Toolbox at the following URL: http://roveri.faculty.polimi.it/software-and-datasets/.

[5]In the experimental analysis in Section 5, $F^{max} = 5$.

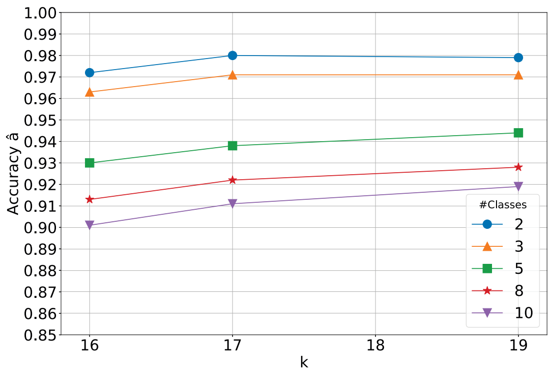
[6]The proposed methodology can be applied to any other Feed-Forward Convolutional Neural Networks. For example, the methodology could be applied to the ResNet [25], where the task-dropping procedure described in Section 3 considers each "Residual block" as a single layer.



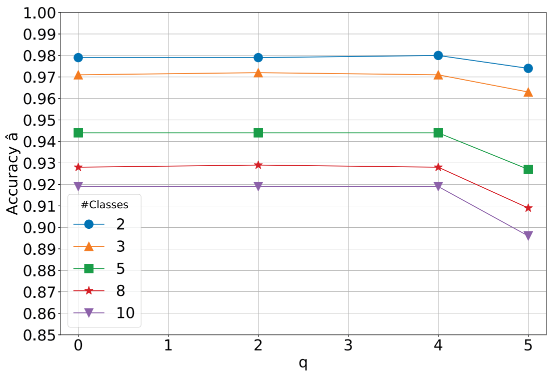
(a) The values of the accuracy $\hat{a}$ for the AlexNet and for the different classification algorithms when k ranges from 16 to 19, $q = 0$, and the number of classes $N^{\tilde{\Lambda}} = 3$.



(b) The values of the accuracy $\hat{a}$ for the AlexNet and for the different classification algorithms when q ranges from 0 to 5, $k = 19$, and the number of classes $N^{\tilde{\Lambda}} = 3$.



(c) The values of the accuracy $\hat{a}$ for the AlexNet and the SVM when k ranges from 16 to 19, $q = 0$, and the number of classes $N^{\tilde{\Lambda}}$ ranges from 2 to 10.



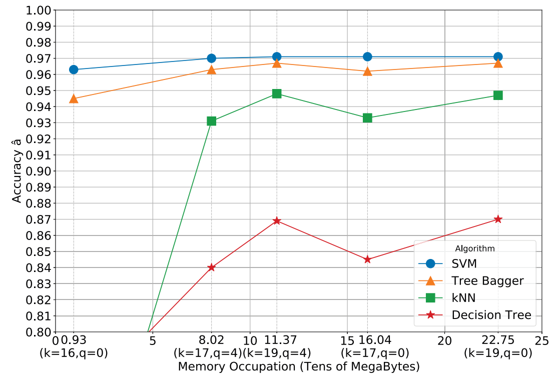
(d) The values of the accuracy $\hat{a}$ for the AlexNet and the SVM when q ranges from 0 to 5, $k = 19$, and the number of classes $N^{\tilde{\Lambda}}$ ranges from 2 to 10.

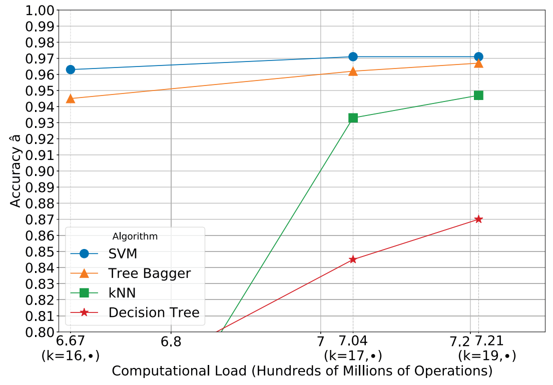
Figure 3: The experimental results showing the values of $\hat{a}$ when k ranges from 16 to 19 and q from 0 to 5 with different learning algorithms and different number of classes $N^{\tilde{\Lambda}}$. Here, the considered CNN is the AlexNet.

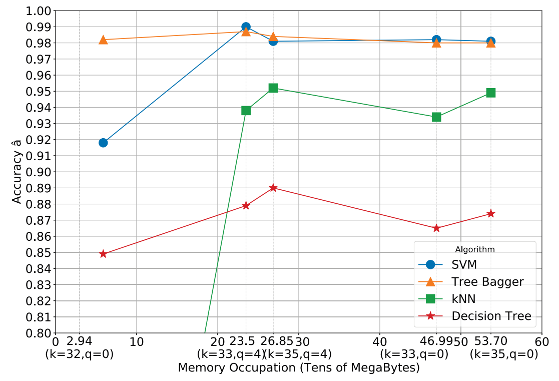
5.1 Synthetic analysis on image recognition

We evaluated the proposed methodology $\mathcal{M}$ on different configurations of k, q, Φ, i.e., AlexNet and VGG-16, and number of classes $N^{\tilde{\Lambda}}$ in T. The specific set of classes in each experiment is randomly extracted from the *Caltech-256*. Then, the settings for each experiment are the following: the number of samples per each class is the minimum between two-hundred and the minimum number of samples in the target categories; these images are then randomly split into a training (70%) and a test set (30%). Experiments are repeated 100 times and results averaged.

The outcomes of this experimental analysis are shown in Figure 3 and 4. More specifically, Figures 3(a) and 3(b) show the values of the classification accuracy $\hat{a}$ when k ranges from 16 to 19 and q from 0 to 5, respectively, with different learning algorithms. Here, the number of classes $N^{\tilde{\Lambda}}$ in T is equal to 3 and the considered CNN is the AlexNet. These results are particularly interesting since they show that applying task dropping and precision scaling has a minimal effect on the classification accuracy. This allows to support the idea of $\widetilde{\Phi}$ as an application-specific and approximated version of Φ. Reducing k and increasing q allows to significantly reduce both memory occupation and computational load in $\widetilde{\Phi}$ as described in Section 4.1. In fact, even considering $q = 0$, reducing k from 19 to 16 allows to reduce the computational


(a) The values of the accuracy $\hat{a}$ w.r.t. $M^{\widetilde{\Phi}}$ for different configurations of (k, q). The considered CNN is the AlexNet and the number of classes $N^{\tilde{\Lambda}} = 3$.


(b) The values of the accuracy $\hat{a}$ w.r.t. $C^{\widetilde{\Phi}}$ for different configurations of (k, q). The considered CNN is the AlexNet and the number of classes $N^{\tilde{\Lambda}} = 3$. Here, q does not affect the computational load.


(c) The values of the accuracy $\hat{a}$ w.r.t. $M^{\widetilde{\Phi}}$ for different configurations of (k, q). The considered CNN is the VGG-16 and the number of classes $N^{\tilde{\Lambda}} = 3$.

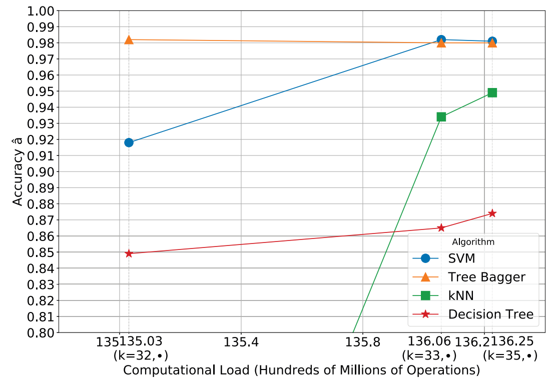

(d) The values of the accuracy $\hat{a}$ w.r.t. $C^{\widetilde{\Phi}}$ for different configurations of (k, q). The considered CNN is the VGG-16 and the number of classes $N^{\tilde{\Lambda}} = 3$. Here, q does not affect the computational load.

Figure 4: The experimental results showing the relationships among $\hat{a}$, $M^{\widetilde{\Phi}}$ and $C^{\widetilde{\Phi}}$ for different configurations of (k, q). The considered CNNs are the AlexNet and the VGG-16, while the number of classes $N^{\tilde{\Lambda}} = 3$. .

load $C^{\widetilde{\Phi}}$ from 721 to 667 million operations and the memory occupation from 227.5 Mb to 9.3 Mb.

Similar results are shown in Figures 3(c) and 3(d), describing the values of $\hat{a}$ for the SVM w.r.t k and q, while the number of classes $N^{\tilde{\Lambda}}$ in T ranges from 2 to 10. Even in this case, results are particularly interesting since they show that the number of classes $N^{\tilde{\Lambda}}$ has a negligible effects in the accuracy reduction when k or q vary. On the one hand, fixing $N^{\tilde{\Lambda}}$, the reduction in $\hat{a}$ is negligible when k decreases or q increases. On the other hand, fixing k or q, the values of $\hat{a}$ depends on $N^{\tilde{\Lambda}}$. Both behaviours are reasonable and in line with what expected.

Experimental results in Figure 4 show the relationship between $\hat{a}$, $M^{\widetilde{\Phi}}$ and $C^{\widetilde{\Phi}}$ for different configuration of (k, q) and for AlexNet and VGG-16. Here, the whole set of classification algorithms has been considered and $N^{\tilde{\Lambda}} = 3$ image classes have been randomly extracted from the *Caltech-256* benchmark. In particular Figures 4(a) and 4(b) show the experimental results about the AlexNet, while Figures 4(c) and 4(d) the ones about the VGG-16. These results show the ability of the proposed methodology to explore configurations of (k, q) to significantly reduce $M^{\widetilde{\Phi}}$ and $C^{\widetilde{\Phi}}$ with a minimal reduction in $\hat{a}$. This specifically holds for the SVM that, for example, provides a reduction from 227.5 MB to 80.2 MB of memory occupation and from 721 to 704 million operations when passing from configuration ($k = 19$, $q = 0$) to ($k = 17$, $q = 4$) for the AlexNet as shown in Figures 4(a) and 4(b). This meaningful reduction in memory occupation and computational load comes at the expenses of a negligible reduction in $\hat{a}$, i.e., less than 1%. Similar comments can be made for results in Figures 4(c) and 4(d) about the VGG-16.

These results corroborate the idea of using $\widetilde{\Phi}$ as an approximated version of Φ to be executed on embedded systems.

5.2 Porting the approximated AlexNet to two off-the-shelf embedded platforms for image recognition

The proposed methodology has then be tested on an image-recognition embedded application whose technological targets are two well-known off-the-shelf embedded platforms: the STM32F7 micro-controller and the Raspberry Pi 3B. Technological details about these two embedded platforms are given in Table 2.

To allow a fair evaluation of the proposed methodology, the considered image-recognition application has been modelled as a two-class classification problem aiming at distinguishing between the class *people* and *car* of the *Caltech-256* benchmark. Hence, the considered dataset T encompasses 209 images of people and 116 of cars.

To ease the comparison, in this analysis, the considered Φ is the AlexNet, while the constraint $\overline{M}$ on the memory occupation has been set to one fifth of the available RAM of the STM32F7. The constraint on $\overline{C}$ has been set to 100.0×10^6.

Given the dataset T and the constraints $\overline{M}$ and $\overline{C}$ for both embedded platforms, the methodology $\mathcal{M}$ has been applied and results are given in Table 2. These results show the ability of the proposed methodology to design effective $\widetilde{\Phi}$ satisfying the constraints given by the technology. In particular, for the STM32F7, the selected configuration of $\widetilde{\Phi}$ refers to $k = 5$ and $q = 0$. In this specific case, the filter-selection optimization described in Section 4.3 has been carried out, leading to the use of just one filter, i.e. $f = 1$ (the selected filter is the tenth filter of first convolutional layer of the AlexNet; the normalization step has not been here carried out). This lead to very low memory occupation (i.e., $M^{\widetilde{\Phi}}$ = 1.4KB) and computational load (i.e., $C^{\widetilde{\Phi}} = 1.09 \times 10^6$). The selected μ_Θ is the Decision Tree, while the estimated accuracy is $\hat{a} = 87.9$. The execution time e_t for the processing of one image by the designed $\widetilde{\Phi}$ is approximately 2700ms (measured through an oscilloscope).

The results about the Raspberry Pi 3B show $\hat{a}$, $M^{\widetilde{\Phi}}$ and $C^{\widetilde{\Phi}}$ for three different configurations, i.e., $q = 0$ and filter-selection, $q = 0$ and no filter-selection, and $q = 4$ and no filter-selection. As expected, when the Raspberry Pi 3B operates in the same configurations of the STM32F7 (i.e., second column of Table 2), the only difference resides in e_t, which is significantly lower for Raspberry Pi 3B thanks to the more powerful CPU. The third column of Table 2 refers to the case where the filter-selection optimization is not carried out leading to higher $M^{\widetilde{\Phi}}$, $C^{\widetilde{\Phi}}$ and e_t but guaranteeing a very high $\hat{a}$. Finally, the results about the configuration without filter-selection optimization but with $q = 4$ are shown in the forth column of Table 2. As expected, this configuration halves $M^{\widetilde{\Phi}}$ with no effect on $C^{\widetilde{\Phi}}$ and e_t[7]. Interestingly, this reduction in $M^{\widetilde{\Phi}}$ does not come at the expenses of $\hat{a}$ that is in line with the one with $q = 0$.

6 CONCLUSIONS AND FUTURE WORKS

Embedded systems have never been considered as a viable technological solutions for CNNs due to their high demand on computation and memory. This paper aims at filling this gap by introducing a methodology for the design and porting of CNNs to embedded systems by employing approximate computing techniques to reduce the computational load and memory occupation. The proposed methodology has been validated on two state-of-the-art CNNs and applied to an image-recognition application running on two relevant off-the-shelf embedded platforms.

Future works encompass the use of the designed Convolutional Neural Networks for Embedded Systems in pervasive distributed systems (e.g., those based on Fog Computing/Networking) where (part of the) computation can be offloaded to high-performance units in a distributed way. In addition, as commented in Section 3, the designed Convolutional Neural Networks for Embedded Systems can natively be part of adaptive systems able to operate in nonstationary environments. In this case, ad-hoc adaptation mechanisms to reconfigure the classification and/or the convolutional layers will have to be designed. Finally, the proposed methodology will be extended to other standard Convolutional Neural Networks frameworks (e.g., Tensorflow, Caffe, etc..).

ACKNOWLEDGMENT

This work has been partially supported by the project "GAUChO - A Green Adaptive Fog Computing and Networking Architecture" funded by MIUR under PRIN Bando 2015.

REFERENCES

[1] Cesare Alippi. 2014. *Intelligence for Embedded Systems: A Methodological Approach.* Springer.

[7]The q parameter mainly affects the memory occupation, while its effect on the execution time is strictly dependent on the specific implementation (and platform): in the hardware platforms we considered the use of 16-bit FP (the simplest implementation of q=4 case) in the C code does not reduce the execution time since the GCC-compiler limits the usage of this data-type to storage purposes only, casting the operands to 32-bit FP during processing.

Table 2: Porting an image-recognition application based on the AlexNet to STM32F7 and Raspberry Pi 3B.

Platform	STM32F7	Raspberry Pi 3B		
CPU	ARM M7 @ 167 MHz	ARM11 @ 1.2 GHz		
RAM	512 KB	1024 MB		
$\overline{M}$	102 KB	102 KB		
$\overline{C}$	100.0×10^6	100.0×10^6		
CNN Φ	AlexNet	AlexNet	AlexNet	AlexNet
k	5	5	5	5
q	0	0	0	4
Filter-selection mechanism	yes	yes	no	no
f	1	1	-	-
μ_Θ	Decision Tree	Decision Tree	SVM	SVM
$\hat{a}$	87.9	87.9	99.3	99.4
$\widetilde{M^{\Phi}}$	1.4 KB	1.4 KB	68 KB	34 KB
$\widetilde{C^{\Phi}}$	1.09×10^6	1.09×10^6	52.7×10^6	52.7×10^6
Exec. Time (ms)	2700	178	8687	8687

[2] Cesare Alippi, Giuseppe Anastasi, Mario Di Francesco, and Manuel Roveri. 2009. Energy management in wireless sensor networks with energy-hungry sensors. *IEEE Instrumentation & Measurement Magazine* 12, 2 (2009).

[3] Cesare Alippi and Pietro Braione. 2006. Classification methods and inductive learning rules: What we may learn from theory. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 36, 5 (2006), 649–655.

[4] Cesare Alippi, Romano Fantacci, Dania Marabissi, and Manuel Roveri. 2016. A Cloud to the Ground: The New Frontier of Intelligent and Autonomous Networks of Things. *IEEE Communications Magazine* 54, 12 (2016), 14–20.

[5] Cesare Alippi, Stavros Ntalampiras, and Manuel Roveri. 2012. An HMM-based change detection method for intelligent embedded sensors. In *Neural Networks (IJCNN), The 2012 International Joint Conference on.* IEEE, 1–7.

[6] Cesare Alippi and Manuel Roveri. 2017. The (Not) Far-Away Path to Smart Cyber-Physical Systems: An Information-Centric Framework. *Computer* 50, 4 (2017), 38–47.

[7] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. 2016. YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* 236–241. https://doi.org/10.1109/ISVLSI.2016.111

[8] Itamar Arel, Derek C Rose, and Thomas P Karnowski. 2010. Deep machine learning-a new frontier in artificial intelligence research [research frontier]. *IEEE computational intelligence magazine* 5, 4 (2010), 13–18.

[9] Christopher M Bishop. 2006. *Pattern recognition and machine learning.* springer.

[10] L. Cavigelli and L. Benini. 2016. A 803 GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology* PP, 99 (2016), 1–1. https://doi.org/10.1109/TCSVT.2016.2592330

[11] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition.* Suvisoft.

[12] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[13] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2012. Implementing neural networks efficiently. In *Neural Networks: Tricks of the Trade.* Springer, 537–557.

[14] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems.* 1269–1277.

[15] Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. 2015. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine* 10, 4 (2015), 12–25.

[16] Aysegul Dundar, Jonghoon Jin, Vinayak Gokhale, Berin Martini, and Eugenio Culurciello. 2014. Memory access optimized routing scheme for deep networks on a mobile coprocessor. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE.* IEEE, 1–6.

[17] Aysegul Dundar, Jonghoon Jin, Berin Martini, and Eugenio Culurciello. 2017. Embedded streaming deep neural networks accelerator with applications. *IEEE transactions on neural networks and learning systems* (2017).

[18] Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. 2010. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on.* IEEE, 257–260.

[19] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. 2014. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops.* 682–687.

[20] Gregory Griffin, Alex Holub, and Pietro Perona. 2007. Caltech-256 object category dataset. (2007).

[21] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15).* 1737–1746.

[22] Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of machine learning research* 3, Mar (2003), 1157–1182.

[23] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[24] Kaiming He and Jian Sun. 2015. Convolutional neural networks at constrained time cost. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 5353–5360.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[26] Robert D. Hof. 2013. 10 Breaktrough Technologies 2013: Deep Learning. *MIT Technology Review* (2013).

[27] IMAGENET. 2017. Large Scale Visual Recognition Challenge (ILSVRC). In *http://www.image-net.org/challenges/LSVRC/.*

[28] Jonghoon Jin, Vinayak Gokhale, Aysegul Dundar, Bharadwaj Krishnamurthy, Berin Martini, and Eugenio Culurciello. 2014. An efficient implementation of deep convolutional neural networks on a mobile coprocessor. In *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on.* IEEE, 133–136.

[29] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[30] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems.* 1097–1105.

[32] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 3431–3440.

[33] Franck Mamalet and Christophe Garcia. 2012. Simplifying convnets for fast learning. *Artificial Neural Networks and Machine Learning–ICANN 2012* (2012), 58–65.

[34] Franck Mamalet, Sébastien Roux, and Christophe Garcia. 2010. Embedded facial image processing with convolutional neural networks. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on.* IEEE, 261–264.

[35] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 62.

[36] Sinno Jialin Pan and Qiang Yang. 2010. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.

[37] Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. 2013. Learning separable filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2754–2761.

[38] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. 2013. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229* (2013).

[39] Pierre Sermanet, Koray Kavukcuoglu, and Yann LeCun. 2009. Eblearn: Open-source energy-based learning in c++. In *Tools with Artificial Intelligence, 2009. ICTAI'09. 21st International Conference on.* IEEE, 693–697.

[40] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[41] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.

[42] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039* (2017).

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 1–9.

[44] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, Vol. 1. 4.

[45] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision.* Springer, 818–833.

[46] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 161–170.

[47] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. 2015. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision.* 1529–1537.