International Journal of Parallel Programming
https://doi.org/10.1007/s10766-020-00654-2

Deep Learning Inference with Dynamic Graphs on Heterogeneous Platforms

V. Pothos[1] · E. Vassalos[1] · I. Theodorakopoulos[1] · N. Fragoulis[1]

Received: 6 August 2019 / Accepted: 31 January 2020

Abstract
One major drawback of deep-learning algorithms is the elevated cost of computing complexity and memory bandwidth required for inference. In order to ameliorate these costs in applications that utilize Convolutional Neural Networks (CNNs), a new, radical, approach is the dynamic pruning of kernels which aims to the parsimonious inference by learning to exploit and dynamically remove the redundant capacity of a CNN architecture. This conditional execution approach formulates a systematic and data-driven method for developing CNNs that are trained to eventually change size and form in real-time during inference, targeting to the smaller possible computational footprint. The conditional execution however, induces a number of challenges when it comes to the implementation of these algorithms to embedded systems. In this paper we present a systematic way of deploying this new dynamic pruning methodology, in heterogeneous platforms that facilitate both CPU and GPU subsystems. Realtime measurements of embedded implementations in modern SoCs verify the efficacy of the proposed methodology and demonstrate the ability of the dynamic networks to both adapt their size to the complexity of the task and deliver significant computational gains during inference.

Keywords Deep learning · Convolutional neural networks · Heterogeneous platforms · Conditional execution · Dynamic pruning

1 Introduction

In recent years, there has been a surge of interest in the potential of Convolutional Neural Networks (CNNs) and ever since they have been established as the dominant technology for tackling real-world, visual understanding tasks. A significant research effort has been put into the design of (very) deep architectures, able to construct high-order representations of visual information. The accuracy obtained by

N. Fragoulis
nfrag@iridalabs.gr

[1] Irida Labs S.A., Patras, Greece

deep architectures on image classification and object detection tasks [1, 2], proved that the depth of representation is indeed the key to a successful implementation.

Although high quality implementations are already available for mainstream, PC-like computing systems, deploying such implementations into diverse technological areas (i.e. automotive, transportation, IoT, medical etc.), requires development of deep-learning architectures on embedded heterogeneous platforms that operate with limited hardware resources and often within a restricted power budget. Authors in [3] present a study of the available deep-learning frameworks, programming models, general implementation limitations as well as real-world performance results on heterogeneous platforms, focusing on mobile phone SoCs and android OS. The scope of this study though, is to represent general implementation guidelines rather than a detailed scheme on how to deploy CNNs on such platforms.

Furthermore, meeting specific performance requirements on embedded platforms is, in general, arduous, while building systems based on existing computing libraries (e.g. BLAS, Eigen etc.), although possible, usually leads to only limited effectiveness, according to the authors' experience. Based on the above discussion it becomes evident that improving such approaches requires tuning multiple computational kernels for the particular use-case at hand, thus requiring great effort and insight in order to be able to tweak—when and if necessary—any given architecture.

Structural plasticity, that is the ability of sparsely connected networks to change their wiring and connectivity patterns, has proven to be a key mechanism of neuronal circuits [4], increasing a network's learning capacity through the expansion of the "effectual connectivity". A form of this mechanism can be integrated into the deep CNN architectural models, the most demanding class of advanced inference algorithms, especially dominant in the field of vision-oriented applications. In CNNs, the computational graph is organized into groups of nodes—called layers—where the main operation is the convolution of an input tensor with a set of kernels with learned weights. The connectivity of the computational graph, and the number of kernels is defined during training and are fixed during inference.

Under this scheme, a form of dynamically altering connectivity can naturally occur, by integrating a mechanism able to decide the number and identity of the kernels that need to be computed during inference, based on the data being processed on each occasion and discard the rest. Intuitively, such mechanism can effectively enable a large number of sub-models to be potentially available, each of which using a subset of the learned kernels and layers. That way, the capacity of the initial model is increased, with apparent benefits to both accuracy and computational parsimony. In fact, such models have proven to be effective for both simple [5] and more challenging applications [6]. Given the intrinsic bottlenecks occurring by the branching operations on the computational flow of models with conditionally executed parts, the question that arises is whether their theoretical advantages can be translated in real time/power gains, when implemented in off-the-shelf heterogeneous platforms.

This paper focuses on some implementation techniques, that they enable efficient porting of the approach explained above to modern heterogeneous SoCs, that are optimized for high throughput and implement highly parallelized processing schemes. Experimental evidence is demonstrated, supporting that popular mobile SoCs featuring a number of CPU cores and GPUs such as Adreno and MALI, on chipsets like Snapdragon 820 and Exynos 8, can actually deliver the predicted performance benefits of this approach when utilized in a heterogeneous setup. The outcome of the experiments indicates that with proper handling, the total overhead induced by factors such as (i) extra operations for identifying the appropriate kernels to be applied for each layer, (ii) memory re-organization, (iii) inter-device communications and branching-related delays, can be limited to ~ 10% of the overall computational time of an equivalent static model. Therefore, the exploitation of a dynamic, conditional graph can be beneficial if it can deliver the same accuracy for a discount over 10% in the computational budget, compared to the corresponding static solution. This is a rather modest goal for such techniques, which even for very challenging problems and compact CNN model architectures, can deliver more than 30% reduction in overall (average) Multiply-ACcumulate (MAC) count compared to the original model, without loss of accuracy [6].

2 Reduction of the Computational Load of a Neural Network

2.1 Static Pruning

The reduction of the computational load associated with a specific deep-learning structure is the enabling factor towards the broadening of the application field of these structures to IoT and in applications featuring a system with low computational capabilities, in general. In this direction, many researchers attempt to exploit the data sparsity and the redundancy of the parameters inherent in CNNs in order to prune some parts of the convolutional network and thus ease the computational load of the overall structure, in an off-line, post-training approach. In some methods, the coefficients of a CNN are analyzed after training and some of them are zeroed according to their magnitude, leading to sparse matrices exploitable by sparse arithmetic software. In some others, the CNN is trained in such a way so to result on a set of coefficients containing as few insignificant coefficients as possible.

In a data-driven approach Hue et al. in [7] proposed a method which iteratively optimizes the network by pruning unimportant neurons based on analysis of their outputs on a large dataset.

Feng et al. [8] proposed a method for estimating the structure of the model by utilizing un-labelled data. Their method called Indian Buffet Process CNN (ibpCNN), captures the distribution of the data and accordingly balances the model between complexity and fidelity.

Similarly, Wen et al. [9] incorporated Structured Sparsity Learning (SSL) in order to regularize the number of filters (and their shapes), the number of channels and the depth of the network. From an implementation perspective, SSL also targets to the formulation of a dense weight matrix in order to completely remove channels, filters or even whole layers.

Yang et al. [10] proposed an energy-aware pruning algorithm for CNNs that directly uses energy consumption estimation of a CNN to guide the pruning process. For each layer, the weights are first pruned and then locally fine-tuned with a closed-form least-square solution to quickly restore the accuracy.

Authors in [11] proposed a three-step method, which allowed them to prune redundant connections without affecting the accuracy. In the first step, they train a network to learn which connections are important. In the second stage, connections characterized as unimportant are pruned and in the last stage, the network is re-trained in order to fine-tune the weights.

Similarly, in [12], authors target implementations for low power devices, by taking advantage of the sparsity immanent in intermediate filter responses in order to reduce the spatial convolution at every layer. More specifically, they are inspired by the loop perforation technique (originally proposed for source code optimization) in order to skip the convolution operation at several locations.

2.2 Dynamic Pruning

Since the main source of computational load in a CNN is the number of convolutional kernels employed in each convolutional layer, one idea proposed in [6] and reviewed here, is to enforce channel-wise sparsity to the outputs of each convolutional layer. In this way, each kernel either learns how to capture useful information or else vanishes. In contrast to the regularization approach [9], which tries to enforce a global sparsity pattern in order to prune kernels and channels with zero output values, we propose a technique that enables kernels to learn information which may be useful to a subset of the observed cases. One step forward, by enforcing this sparsity via simultaneously learned, data-driven, kernel activation rules, the same rules can be used during inference in order to avoid computing kernels which are not useful for a particular datum. That way, only the relevant kernels are computed, resulting to a significant economy in processing time and power. At the end of the training procedure, kernels that have not managed to learn features that are relevant to any of the data, resulting to zero utilization, can be permanently pruned from the model.

2.3 The Learning Kernel-Activation Module

Figure 1 depicts *i-th* and the $(i+1)$-*th* convolutional layer of a typical convolutional network and introduces the Learning Kernel-Activation Module (LKAM). The LKAM links two consecutive convolutional layers and acts as learning switch, capable of switching on and off individual kernels of any layer depending on its input, that is, the output of the previous convolutional layer.

The module learns which kernel to disable during the CNN training process, which is for that reason specifically devised to facilitate such operation by exploiting data sparsity usually employed in images.

The aim of these modules is initially to induce the desired channel-wise sparsity to the feature maps. Simultaneously, they learn an activation rule for each kernel, which is later been exploited during the inference phase. Many types of activation rules can be formulated using regular differentiable functions, available in all deep-learning frameworks. In this work one of the simplest and lightweight rules is studied, constituting by a bank of 1×1 convolutional kernels followed by average pooling and a sigmoid function offering a smooth and differentiable transition

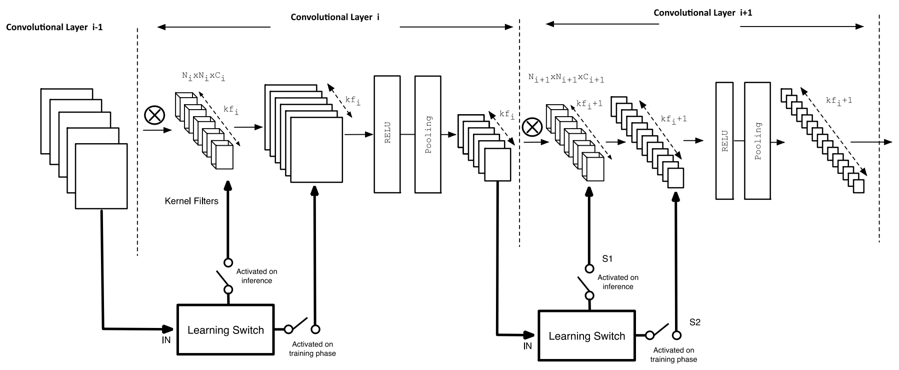

Fig. 1 The learning switch module is introduced between successive convolutional layers of a CNN

between active and inactive state. The choice of this rule was made in order to keep the computational overhead of the LKAM modules as low as possible. The internal structure of the LKAM module is shown in Fig. 2. First, the feature maps of the *i-th* convolutional layer are fed into this module. These are processed by $k_{fi} + 1$ kernels of size $1 \times 1 \times C_i + 1$ resulting into $k_{fi} + 1$ feature maps. These maps are then fed into a global average-pooling block, which averages the values of each map producing a corresponding single number.

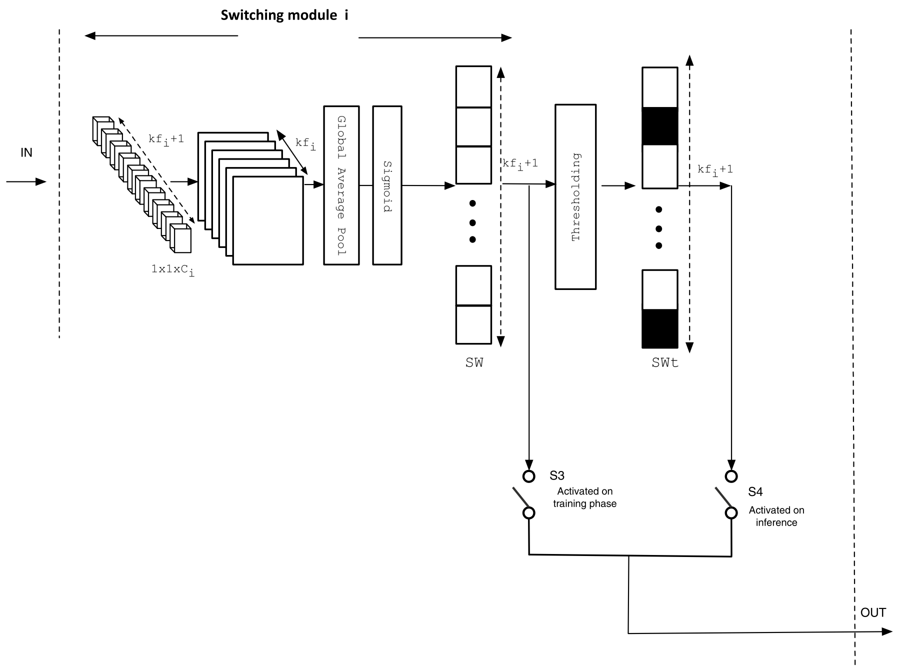

Fig. 2 The internal structure of a LKAM module

Each one of this numbers is then passed into a sigmoid layer implementing the following function:

$$f(x) = \frac{1}{1 + e^{-k(x-x_0)}} \tag{1}$$

In this way a vector $SW = \left\{ sw_1, sw_2, \dots, sw_{k_{fi+1}} \right\} \in \mathbb{R}^{k_{fi}+1}$ with values between 0 and 1, is formed.

The elements of this vector are used in the training phase, through the switch S3, in order to multiply the values of the corresponding feature map in the $(i+1)$-*th* convolutional layer, thus imposing the desired sparsity. During this phase, switches S2 in Fig. 1 and S3 in Fig. 2 are activated, while switches S1 in Fig. 1 and S4 in Fig. 2 are deactivated. This way, the information flow is tweaked by enforcing certain feature maps to gradually have smaller influence on the overall network under the corresponding rules, which are in turn co-adapting. The goal of the training process is to obtain the combination of kernels and activation rules that produce the sparsest *SW* vectors possible. The learned rules can indicate the kernels with zero influence so as to be excluded from further computation. These invalidated kernels will eventually lead to excluding the respective CNN channels from the overall computation load, resulting in both less MAC operations and significant speedup in inference timings.

2.4 Real-Time Deactivation of Kernels During Inference (Recognition) Phase

During *inference*, the elements of the vector *SW* are used as a set of switches that control the corresponding kernels in the *(i + 1)-th* convolutional layer, depending on the input from the *i-th* layer (Fig. 3). Since the value of each sw_i can be any real number between 0 and 1, a simple thresholding is used as the activation criterion, where the elements of the vector *SW* are binarized (i.e. forced to take values 1 or 0) using a threshold value, *thres*, as follows:

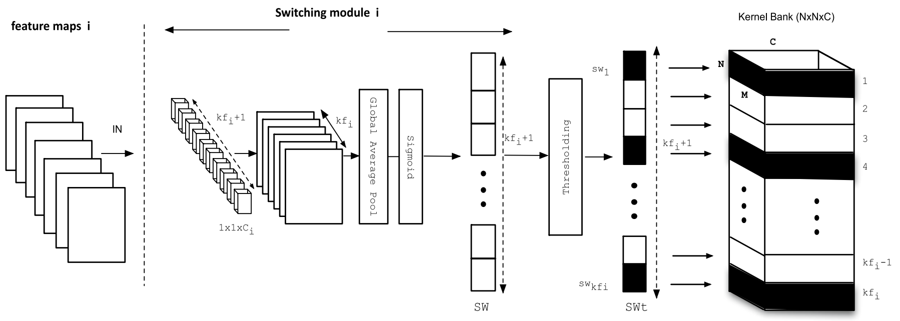

Fig. 3 The switching process of the LKAM module during inference

$$sw_i = \begin{cases} 0, sw_i < thres \\ 1, sw_i \geq thres \end{cases} \tag{2}$$

The resulting binary activation values are the indicators of whether to apply the corresponding filtering kernels on the input data or to skip these particular computations. Note that during inference, switches S2 in Fig. 1 and S3 in Fig. 2 are considered active while switches S1 Fig. 1 and S4 in Fig. 2 are considered inactive.

3 Implementation Considerations of Dynamic Pruning

When the above-mentioned dynamic pruning scheme is applied over a convolutional layer, fewer convolutional kernels are eventually used and the resulting feature map consists of a three-dimensional tensor which has several invalidated channels. These invalidated channels, that correspond to the kernels that were not used in the convolution, should be excluded from any operation of any following-up layer that is using their data. For network layers that may need to operate with data that are directly related with the excluded channels, it's important to have access to the information of which channels have been excluded. That essentially means that Dynamic Pruning requires considering pruning information transmission among network layers as well. This information is usually an array of indices of the valid channels.

In this paper the focus in on Convolutional and Full Connected layer CPU/GPU implementations since those layers contribute the most to the overall computational complexity of the CNN.

3.1 Implementation of Convolutional Layers

The implementation of a convolutional layer is based on General Matrix Multiplication (GeMM) algorithms. GeMMs consist a well-studied problem and several GeMMs implementations has been incorporated into a lot of highly optimized libraries. Such implementations realize the data prefetching/caching, vectorization and threading mechanisms, that do exist in modern heterogeneous systems, very efficiently which in turn leads to lower latency and increased processing performance.

In general, the input feature maps of the convolutional layers need to be transformed properly in order to be used with GeMMs. The required preprocessing transformation is achieved via algorithms that convert the three-dimensional tensors to two-dimensional arrays, properly designed for GeMM processing. These transformations, namely *im2col* (image-to-columns) or *im2row* (image-to-rows), come at the costs of preprocessing speed and increased memory footprint as they do replicate the data. Their preprocessing speed impact, however, compared to the following up GeMM algorithms, is often negligible.

Assuming that the GeMMs algorithms are provided as optimized "black-box" libraries, meaning that the end user has no access to view and modify their implementation, the *im2col* pre-processing step should incorporate all the necessary changes to accommodate the dynamic pruning throughout the whole convolution process.

Given the predefined size of the kernels that are used in a convolution operation, the *im2col* algorithm is designed to create a column with the corresponding input tensor data that should be convolved with the kernel in order to result a number for the output tensor. Supposing that there are K_n kernels of dimensions $\left[K_c, K_h, K_w\right]$, then the *im2col* algorithm should produce, for every expected output, a column with length equal to $K_c \times K_h \times K_w$. Supposing that the output of the convolution should be a tensor of dimensions $\left[K_n, H, W\right]$, then this can be achieved by the GeMM operation as provided in Eq. 3 and depicted graphically in Fig. 4.

$$[K_n, K_c \times K_h \times K_w] \times [K_c \times K_h \times K_w, W \times H] = [K_n, W \times H] == [K_n, H, W] \quad (3)$$

The *inactive kernels* can be easily handled. They should not be included in the left matrix of the GeMM formula (3) which corresponds to the gray parts of the Fig. 4 that can be omitted. This means that the kernels array $\left[K_n, K_c \times K_h \times K_w\right]$ should be replaced with the array $\left[K_a, K_c \times K_h \times K_w\right]$ where $K_a \leq K_n$ is the number of active kernels for the convolution. For CPU-based implementation this change will require a proper copy of the coefficients of the initial kernels to an intermediate memory buffer according to the corresponding coefficients of the active kernels. In the worst case scenario, where $K_a = K_n - 1$, almost a full copy of the coefficients in the intermediate memory will be required (note that if $K_a = K_n$ there is no need for intermediate buffer). The GeMM algorithm will then produce a tensor of shape $\left[K_a, H, W\right]$.

For the *excluded input channels*, further actions should be taken on the algorithm implementation, such as the exclusion of these channels, by the *im2col* algorithm, at its output. Since in the above-mentioned approach the output of dynamic pruned convolutions will only contain the activated channels in a shape $\left[K_a, H, W\right]$, the *im2col* algorithm eventually will not have to deal at all with the inactive channels. However, the inactive channels should still be removed from the coefficients of the convolution kernels. If the kernels have initial shape

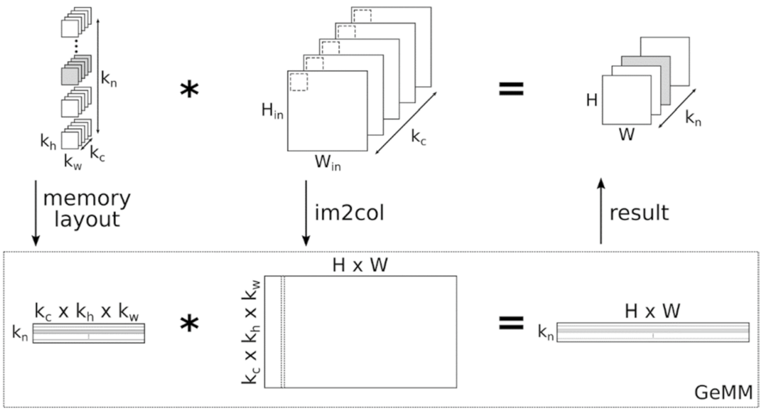

Fig. 4 Convolution with im2col and GeMM algorithms

$[K'_n, K'_c \times K'_h \times K'_w]$, there are $K_a \leq K_n$ active input channels and $K'_b \leq K'_n$ active kernels for the convolution. Hence, a proper copy of coefficients should take place to result into coefficients of shape $[K'_b, K_a \times K'_h \times K'_w]$. Then, the GeMM will produce the expected result as provided by the Eq. 4. A graphical depict of this operation is given at Fig. 5, where two out of five inactive channels have been excluded due to a previous dynamic pruning operation (light gray tone). Although *im2col* algorithm will automatically ignore those excluded (non-existed) channels, proper care must be taken in the first part of the GeMM operation thus to exclude also the coefficients of the convolution that corresponds to the excluded channels (black tone). At the same time inactive kernels (gray tone) are taken into consideration as analyzed before (Fig. 4).

$$[K'_b, K_a \times K'_h \times K'_w] \times [K_a \times K'_h \times K'_w, W' \times H'] = [K'_b, W' \times H'] == [K'_b, H', W'] \tag{4}$$

The above procedure, described for CPU-based implementations, can be in general also used for GPU-based implementations as well. However, GPUs offer the ability to employ more efficient approaches avoiding issues such as data preprocessing, especially if they require a lot of GPU memory (e.g. for *im2col* transformations) or extra copying of coefficients to intermediate buffers.

The key to understand GPU OpenCL-based implementation is to understand the notion of global and local workgroups (WGs) and workitems. A workitem (WI) it can be seen as a thread that executes a block of code in parallel to other WIs. Several WIs forms a local WG (LWG) and inside the LWG, the WIs can have access to the same L1 cache and the local shared memory (LDS, if supported) which is relatively small but very fast. Typically, LDS is used to enable coalesced accesses, to share data among the WIs in an LWG and reduce accesses to global memory which is of lower bandwidth. Each WI inside the LWG can be identified by unique local IDs that are available via built-in functions (i.e. *get_local_id()*). It must be noted that, in general, not all the WIs inside an LWG are executed in parallel but tends to be divided in sub-groups (wavefronts/warps) of

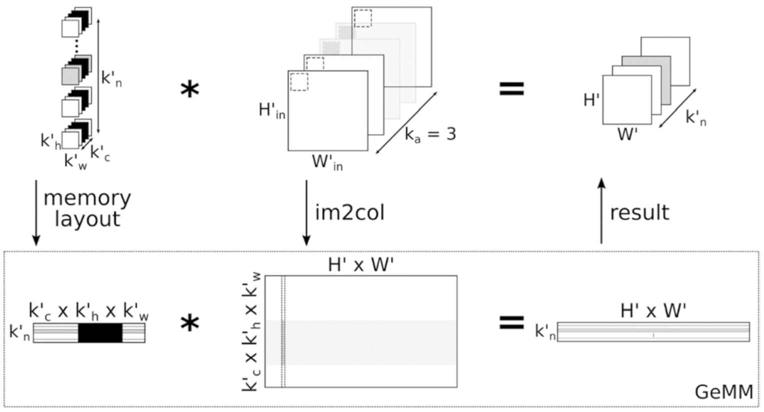

Fig. 5 Convolution with excluded input channels using im2col and GEMM

WIs that are indeed executed in parallel. GPU scheduling mechanisms continuously and properly activate the warps in the LWG, and among different LWG if it's possible, in order to hide latencies.

The size of a warp is important because if the WIs of the same warp are forced to execute different code (for example via -if statements) all code-diverging paths will be executed and the outcome will be masked in the end, increasing this way the processing time. Each LWG is running on a Compute Unit on the GPU and all the LWGs forms the global workgroup (GWG). That means that each LWG can be identified by unique group IDs using the built-in functions (i.e. *get_group_id()*). As all WIs are executing the same OpenCL function, the function's code must utilize the local and group IDs in order to properly process different kind of data. The local and group IDs in OpenCL are expressed as a 3D notion i.e. they are described by three different indices. Hence, each WI is identified by three local ids and three group ids. The above description is depicted visually in Fig. 6.

Under the above design scheme of accessing and programming the GPUs, it becomes obvious that the algorithms implementation can be quite different compared to the CPUs. An approach to implement GPU algorithms is to start by mapping regions of output data to LWGs, trying to avoid multiple utilization of common input data among the different LGWs. The output should be also selected in such way for the input to be cache friendly. Knowing the number of Compute Units of the hardware, typically the number of LWGs should be a multiple of that number in order to utilize at maximum all Compute Units.

In all our test-designs the GWG is designed to have three dimensions. In most of the cases each dimension is related with a specific operational part of the convolution algorithm as depicted in Fig. 7. More specifically:

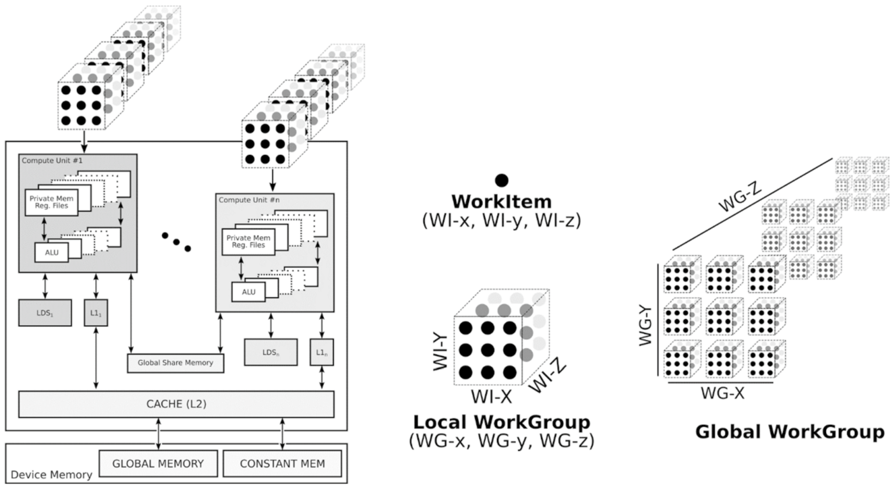

Fig. 6 GPU and WorkGroups/WorkItems interrelationship

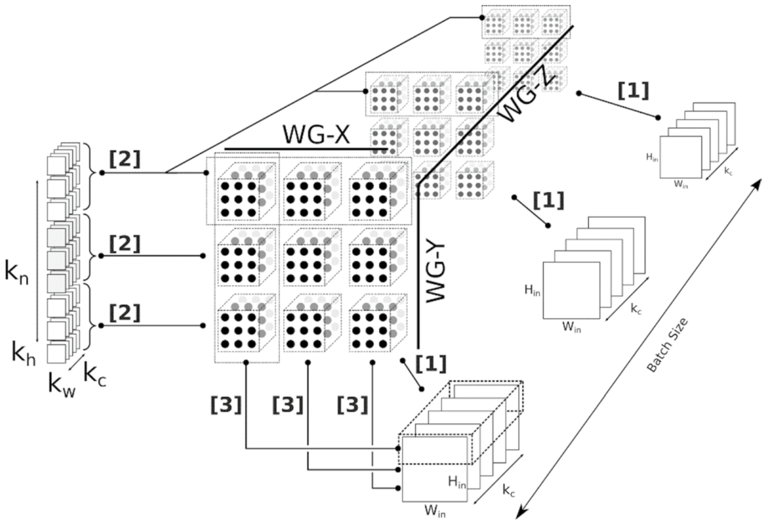

Fig. 7 Graphical example of the correspondence between GWG dimensions with the convolutional kernels and input data processing

1. One dimension is always referring to the batch processing, i.e. is related to the image index that is processed during a batch processing. Based on this dimension the input/output memory address pointers are changed accordingly.
2. Another dimension is usually referring to a number of convolutional kernels that the LWG will process.
3. The remaining dimension is referring to the y-index of a block of an image that is processed by the LWG.

The order of the index of the GWG dimensions in correspondence to the operational parts of the convolution, in most of the cases does affect the processing speed, as it's closely related to the order of the executed LWGs in each compute unit and thus the way of using the intermediate caches. Therefore, different combinations of indexing between of the GWG dimensions and the corresponding operation parts of the code need to be evaluated in order to select the optimal one.

In some cases, however, where for example a convolution layer does have grouping, further dimensions can be used, encapsulated into the three dimensions. For example, if the local id in the 0 dimension can take values $lid0 \in [0, 64)$ then it can produce two new local ids:

- $lid0a = lid0 \div 32$: thus $lid0a \in [0, 2)$ and can be used as group identifier for the 2 groups
- $lid0b = lid0\%32$: thus $lid0b \in [0, 32)$ and can be used as WI index per group for the 2 groups

For GPUs that support LDS memory inside the LWGs, custom code is written to use as much WIs as possible in order to load reusable data in the LDS memory. Zero padding cases are also handled in this step by loading the data to specific position of the LDS memory, having first filled once all that memory with zeros. Memory synchronization usually occurs afterwards, unless the WIs that loads specific local memory parts utilize only these parts later.

Further optimization involves the implementation of direct convolution operators by using vectorized types (e.g. float4), broadcasting mechanism (e.g. first operand of multiplication is common for many WIs and the load is broadcasted fast from local memory), and registers for accumulation. This happens iteratively, among input channels for any group of filters that each WI is responsible to calculate. At the end a Rectified Linear Unit (ReLu) is applied using a vectorized build-in max function and the partial results are stored in output memory. In cases where the convolutional kernels do not have unitary spatial size, or in cases where padding or different than one stride is required, the source code can become quite complex.

It should be noted that a key difference between the GPU-based and CPU-based implementations, is that all copies of coefficients to temporal buffers are completely avoided. Instead the indexes of the active kernels are held in a buffer and only the WIs that work on those active kernels are activated and load properly the coefficients from the difference memory addresses. GPU's prefetching mechanisms come handy at this point as they hide well the latency of loading sparse memory addresses (non-consecutive convolutional kernels in memory) by consecutive WIs.

3.2 Implementation of Full Connected Layers

Common implementations of Full Connected (FC) layers, follow a Vector-to-Matrix multiplication, a task that can be directly and efficiently handled by most of the GeMM algorithms. For conditional computing implementation, the only occasion that GeMM cannot properly handle directly is when the output of a convolution layer must be interpreted as a one-dimensional array for the following FC layer. This case is usually implemented via an additional "*flatten*" or "*reshape*" layer. As in the case of the Convolutional layers the coefficients of the FC layer that corresponds to excluded channels must be excluded from the GeMM operation too. However, since the coefficients of the Full Connected layers are stored with a notion of two-dimensional tensors, instead of four-dimensional arrays, the correspondence of the excluded input channels to the coefficients of the Full Connected columns must be taken into consideration in order to exclude them correctly from the GeMM operation. The exclusion of the coefficients is achieved by making use of a coefficients buffer where only the coefficients that should take place in the GeMM operation are stored.

From all the above it becomes evident that, for Dynamically Pruning Convolutional and FC layers in CPU-based implementations using the GeMM approach, there is a need of memory copies of the coefficients of the kernels to intermediate buffers. These memory copies introduce an overhead in the overall performance that in the worst case can lead to almost loading and storing one additional time the coefficients of the network per inference. However, if the number of dynamic activated kernels is not very high, depending the convolutional parameters, the gains on processing speed/power can be quite significant.

In the case of GPU implementation, where the FC layer is realized as a vector to matrix multiplication, an approach is to use each LWG to calculate the result of a few kernels. By testing various implementations, the authors found that the optimal way for achieving this is to iteratively use a number of N WIs to load consecutively N input data and apply multiply-accumulation over a number of M FC kernels, holding the results in M registers. This approach is cache friendly and does not require vectorized types. As in the CPU-based implementations, is important to know which coefficients corresponds to which active channels. This requires an indexing array that maps the input tensor's channels (before flattening) to corresponding active kernels. Then, the procedure of the Full Connected layer is similar to non-dynamic pruning case, with the exception that coefficients should be loaded from indexed memory addresses. If the dynamic pruning procedure activates all the kernels, the indexing mechanism can introduce latency of as large as 25% of the total processing time. However, this indexing mechanism can lead to performance gains for each inactive kernel. As an example, for a multiplication with dimensions [256, 4096] × [4096, 1] the processing time without the indexing mechanism takes roughly 12 ms in a MDP820's GPU. When the indexing mechanism is engaged, the processing time for the same multiplication is increased to 15.5 ms. For the dynamic pruning case the number of the active kernels can be considerably smaller. For example, in a case where only 47 out of 256 kernels are activated, the operation corresponds to a multiplication with dimensions [47, 4096] × [4096, 1] and in that case exhibits a processing time of just 3.5 ms. According to authors experiments, this is a general trend, leading also to comparative performance gains for the overall network model.

4 Experimental Results

Our analysis has been conducted in two different classification problems: (a) A food recognition problem, utilizing the FOOD-101 database [13] comprised of images of food, organized into 101 categories and (b) a general image recognition problem utilizing the ImageNet ILSVRC 2012 [2] dataset comprised of images organized into 1000 categories. These two datasets have been chosen as two classification cases featuring different qualities. In particular, FOOD-101 is considered as a less complex classification case compared to the ILSVRC 2012 dataset due to the smaller number of classes but featuring more abstract visual attributes since food styling can be very diverge. On the other hand, ILSVRC 2012 dataset contains images mainly depicting a single structured object, but similar classes are often discriminated by very fine details.

In the same spirit, two popular CNN architectures have been evaluated: CaffeNet and SqueezeNet 1.1 [14]. CaffeNet is almost identical to AlexNet [1] architecture, being a conventional "vanilla" architecture with a medium-sized parameter space but relatively shallow. On the other hand, SqueezeNet is a deeper architecture incorporating more complex architecture, and although consisting of 50 times less parameters compared to a CaffeNet, it has similar classification performance. The study of these two architectures that exhibit vastly different characteristics in terms of redundancy, is used to highlight the ability of the proposed framework to achieve the required computational parsimony under very different circumstances, by being responsive to the complexity of the data and also to the complexity of the model.

4.1 Recognition Accuracy

The recognition accuracy obtained by training the two architectures under the proposed framework is summarized in Table 1. The threshold for the activation of kernels is 0.5 and the classification accuracy was measured on the validation set for ILSVRC and the test set for FOOD-101. The accuracy on the ILSVRC is compared to the reference baseline models for CaffeNet and SqueezeNet1.1 available on-line.

It is evident that the introduction of an objective towards computational economy has not degraded the obtained accuracy on either of the tested architectures. On the contrary, we observe a notable improvement of the classification accuracy compared to the reference models, on both datasets and both architectures. This reveals the dynamic of the approach, regarding the overall control of the functionality of these CNNs.

4.2 Computational Load

The most important aspect of the presented framework though, is the improvement on the required computational load during inference. A detailed analysis of these two configurations is presented below.

Evaluation on CaffeNet has been contacted for both ILSVRC 2012 and FOOD-101 datasets.

Engagement of the LKAM modules results on the reduction of the kernel filtering operations. This is shown graphically in Fig. 8a, b through the respective kernel activity profiles. In these plots, the vertical axis corresponds to the activation frequency of a particular kernel throughout the validation set, while the horizontal axis corresponds to the kernels of each layer, sorted by ascending utilization (from left to right). For visualization purposes, the horizontal range is normalized and equal for all layers, even though they accommodate different population of kernels. In such

Table 1 Recognition accuracy for the CaffeNet and SqueezeNet models on the ILSVRC 2012 and Food101 datasets

Recognition Accuracy (%)	ILSVRC 2012				FOOD-101			
	Top1	Diff.	Top 5	Diff.	Top1	Diff.	Top 5	Diff.
AlexNet-Conv	57.27	–	80.62	–	68.54		88.44	–
AlexNet- PI	58.46	+1.19	81.21	+0.59	68.86	+0.32	88.61	+0.17
SqueezeNet 1.1-Conv	57.59	–	80.44	–	65.65		86.87	–
SqueezeNet 1.1-PI	59.59	+2.0	82.05	+1.61	67	+1.35	88.04	+1.17

PI stands for Parsimonious Inference approach (this paper)

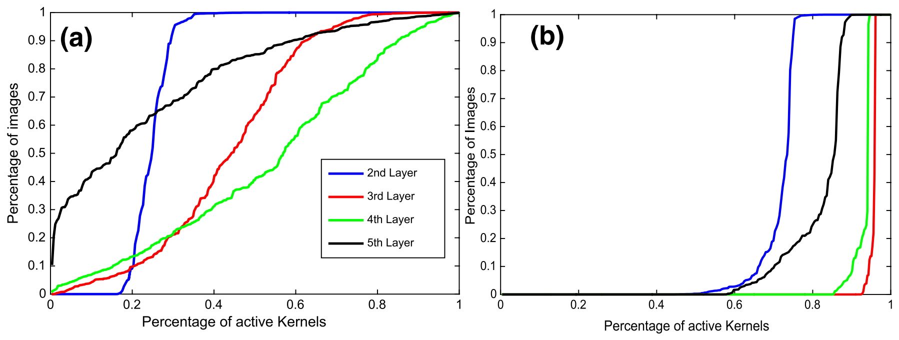

Fig. 8 Kernel Activity Profile for CaffeNet: **a** ILSVRC 2012 dataset, **b** FOOD-101 dataset. A large part of kernels remains permanently inactive

an illustration, a step-like plot implies kernels that are mostly permanently switched either off or on, while a smooth curve indicates kernels whose operation is data-dependent. More specifically, the number of kernels that are calculated for each test image is significantly lower than the nominal number of kernels in the original networks. This dramatically reduces the related number of mathematical operations. It must be noted again that the reduction on the active kernels in a single layer, besides the obvious benefit of avoiding the corresponding filtering computations, results into a respective reduction in the number of input channels into the next layer. This in turn, offers an additional computational gain, directly proportional to the number of switched-off kernels.

A statistical analysis on the switching activity of the CaffeNet for ILSVRC 2012 dataset reveals that only 64.27% of the network's kernels are active (on average) throughout the validation set (35.73% reduction). Specifically, 75.55% of the layer 2 kernels, 57.03% of the layer 3 kernels, 46.93% of the layer 4 kernels and 77.58% of the layer 5 kernels are, on average, activated.

The reduction of the computational load, in terms of the total MACs operations, has been computed to be at 38.31% compared to the reference CaffeNet, taking also into account the computational overhead introduced by the switching modules.

The same analysis on the FOOD-101 dataset indicates that throughout the validation set, only 14.52% of the kernels are activated in the layers of the network. Specifically, 28.49% of the layer 2 kernels, 4.55% of the layer 3 kernels, 6.91% of the layer 4 kernels and 18.15% of the layer 5 kernels are activated on average. Evaluation on SqueezeNet 1.1 has also been conducted on ILSVRC 2012 and FOOD-101 datasets. On the ILSVRC 2012 dataset, the engagement of the LKAM modules results again to the reduction of the kernel filtering operations. This is shown graphically in Fig. 9a, b. A statistical analysis on the switching activity of the SqueezeNet 1.1 for the ILSVRC2012 dataset reveals that on average throughout the validation set, only 68.28% of the kernels are active in the layers of the network leading to a 31.72% reduction.

The results for both CaffeNet and SqueezeNet networks clearly suggest that the proposed architectural modification results into a significant reduction of the active

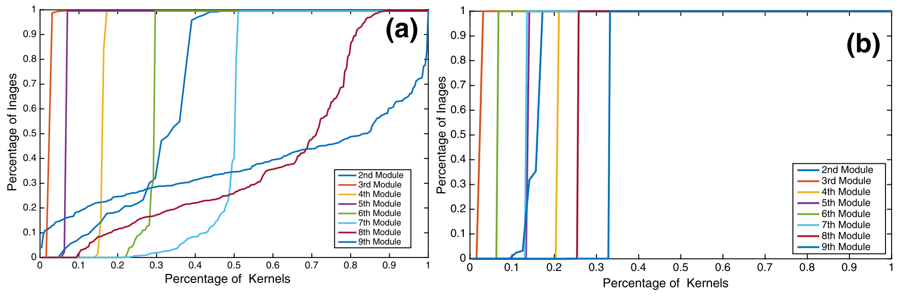

Fig. 9 Kernel Activity Profile for every layer (fire module) of SqueezeNet 1.1: **a** ILSVRC 2012 dataset, **b** FOOD-101 dataset. For the easier FOOD-101 problem kernels are mostly either permanently active or inactive

kernels during inference time. Of equal importance is the fact that the networks adapt their form and size, depending on the data and the complexity of the classification problem: CaffeNet demonstrates a decrease in the average number of active kernels necessary for carrying out the recognition on the FOOD-101 dataset compared to that of the ILSVRC2012 dataset, regardless of being trained with the same regularization gains. That means that the corresponding network features an excess learning capacity, not needed for carrying out this classification task, having got recognized as such and automatically eliminated by the presented training scheme.

4.3 Embedded Implementation and Inference Speed Measurements

The validity of the proposed dynamic pruning, parsimonious, approach has been verified for the CaffeNet and SqueezeNet v1.1 architectures on four different platforms, namely, the Intrinsyc MDP-820 (Qualcomm Snapdragon 820 with Kryo ARM/Adreno 530 GPU), the Xiaomi Redmi Note 4 (Mediatek MT6797 Helio X20 with ARM/Mali-T880 MP4 GPU), the LG-G4 (Qualcomm SnapDragon 808 with ARM/Adreno 418 GPU) and the Samsung S7 Edge (Exynos 8890 Octa with ARM/MALI-T880 MP12 GPU) and the inference speeds of the above were compared with the respective baseline models on the ILSVRC 2012 and FOOD-101 datasets and by using different deep-learning frameworks such as Caffe, Caffe2, TensorFlow, Neural SDK [15] and DeepAPI (*deep-learning library/framework*) developed by the authors [16].

On the above-mentioned experiments the inference computations were either a) assigned entirely to the respective CPU, b) offloaded to the GPU while CPU is mostly used for housekeeping functions, or c) partitioned between the CPU and the GPU. Table 2 depicts the timing performance of the a MDP820-based, parsimonious inference (PI) approach in the case of SqueezeNet v1.1(SQNet1.1) and CaffeNet CNNs, on the FOOD-101 dataset, for specific combinations of CPU and GPU batch sizes, indicating overall speedup of $\times 1.3$–$\times 2.0$ times is achievable, against the conventional baseline CNN implementation, when the proposed, conditional execution approach is deployed. Table 3 includes the acquired mean inference time measurements for the two CNN architectures under considerations for both baseline and PI

Table 2 Inference speed of SQNet1.1/CaffeNet FOOD-101 datasets

Device	Network	CPU batch	GPU batch	Baseline time/ image (ms)	PI time/ image (ms)	Speedup (times)
MDP820	SQNet1.1	12	–	34.1	17.0	×2.01
		–	12	22.6	14.0	×1.61
		8	12	16.2	12.9	×1.26
	CaffeNet	8	–	86.0	58.7	×1.47
		–	12	53.8	32.6	×1.65
		4	6	42.4	27.8	×1.53

implementations, developed under different frameworks and running on various devices and hardware (CPU and/or GPU or DSP).

In all these platforms the GPU is programmed in OpenCL, using hand-optimizations aiming to avoid any pre- and post- processing operations at convolutions layers, minimize memory usage by avoiding temporary memories, reduce as much as possible the data transfers from/to GPU and efficiently exploit the de-activation of the kernels.

From Table 3, it is evident that the implementations derived from the proposed PI methodology lead to faster inference times due to the economy in the computations of filtering kernels while not jeopardizing the CNNs accuracy. It must be noted however that the inference speed-up although proportional to the reduction of the total MACs, is not equal to that. For example, for the GPU implementation of CaffeNet/ILSVRC2012 in Samsung S7 Edge the speed up gain is about 28.3%, while the corresponding MACs reduction is 38.31%. This is because of the computational overhead related to the real-time restructuring of the data and kernel tensors within the GPU and prior to the computations, which is necessary for accommodating kernel switching capabilities.

5 Conclusions

Recently a new CNN design approach has been proposed which allows a CNN to learn to use as few computing resources as possible, and conditionally execute specific filtering kernels, changing its size and form in real-time during inference, depending on the input data.

The proposed framework incorporates a new learning module, the Learning Kernel Activation Module (LKAM), able to dynamically activate or de-activate a subset of filtering kernels (and the corresponding channels), depending on the input image content during inference phase. Using this new module, the CNN learns during the training phase how to reduce its size in real-time and thus to result in a significant computational economy.

The conditional execution however employs a number of challenges when it comes to the implementation of these algorithms to embedded systems. Hence, in this paper we presented a systematic way of deploying this new dynamic pruning methodology

Table 3 Mean inference speeds of CaffeNet/SQNet1.1 on ILSVRC 2012 and FOOD-101 datasets (batch 12) for baseline and parsimonious inference

Device	CNN model	Dataset	Implementation type	Implementation framework	Mean inference time (ms)			
					CPU	GPU	CPU + GPU	DSP
MDP820	SQNet v1.1	Food-101	Baseline	Caffe	270.0	–	–	–
				Caffe 2	219.0	–	–	–
				Tensor flow	390.0	–	–	–
				SNPE-PI	71.7	16.3	–	22.3
				DeepAPI	28.3	21.4	16.2	–
			Parsimonious	DeepAPI	17.0	14.0	12.9	–
	CaffeNet		Baseline	SNPE	128.3	34.4	–	87.6
				DeepAPI	85.5	55.5	42.4	–
			Parsimonious	DeepAPI	58.7	30.2	27.8	–
		ILSVRC2012	Baseline	DeepAPI	–	54.9	–	–
			Parsimonious	DeepAPI	–	50.7	–	–
LG G4	SQNet v1.1	Food-101	Baseline	DeepAPI	96.4	112.1	73.2	–
Xiaomi Redmi Note 4	CaffeNet	Food-101	Baseline	DeepAPI	–	250.2	–	–
			Parsimonious	DeepAPI	–	107.1	–	–
		ILSVRC2012	Baseline	DeepAPI	–	250.7	–	–
			Parsimonious	DeepAPI	–	190.0	–	–
Samsung S7 Edge	CaffeNet	Food-101	Baseline	DeepAPI	–	110.0	–	–
			Parsimonious	DeepAPI	–	36.4	–	–
		ILSVRC2012	Baseline	DeepAPI	–	110.0	–	–
			Parsimonious	DeepAPI	–	78.9	–	–

to implement CNN variants, in heterogeneous platforms that facilitate both CPU and GPU subsystems and we have discussed and presented specific implementation considerations and alternatives.

Realtime measurements of embedded implementations in modern SoCs verify the efficacy of the proposed methodology and demonstrate the ability of the resulting networks to adapt their size to the complexity of the classification task.

Funding This work has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 780788.

References

1. Russakovsky, O., Deng, J., Su, H., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. **115**, 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
2. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems (NIPS), pp. 1097–1105 (2012)
3. Ignatov, A., Timofte, R., Chou, W., et al.: AI Benchmark: running deep neural networks on android smartphones. https://arxiv.org/abs/1810.01109 (2018). Last Revised 15 Oct 2018
4. Knoblauch, A., Körner, E., Körner, U., Sommer, F.T.: Structural synaptic plasticity has high memory capacity and can explain graded amnesia, catastrophic forgetting, and the spacing effect. PLoS ONE **9**(5), e96485 (2014). https://doi.org/10.1371/journal.pone.0096485
5. Bengio, E., Bacon, P.L., Pineau, J., Precup, D.: Conditional computation in neural networks for faster models. https://arxiv.org/abs/1511.06297 (2015). Last Revised 7 Jan 2016
6. Theodorakopoulos, I., Pothos, V., Kastaniotis, D., Fragoulis, N.: Parsimonious inference on convolutional neural networks: learning and applying on-line kernel activation rules. https://arxiv.org/abs/1701.05221 (2017). Last Revised 31 Jan 2017
7. Hu, H., Peng, R., Tai, Y.-W., Tang, C.-K.: Network trimming: a data-driven neuron pruning approach towards efficient deep architectures. https://arxiv.org/abs/1607.03250 (2016). Submitted on 12 Jul 2016
8. Feng, J., Darrell, T.: Learning the structure of deep convolutional networks. IEEE Int. Conf. Comput. Vis. (ICCV) **2749–2757**, 1135–1143 (2015)
9. Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H.: Learning structured sparsity in deep neural networks. In: Proceedings of Advances in Neural Information Processing Systems (NIPS), pp. 2074–2082 (2016)
10. Yang, T.J., Yu-Hsin, C., Vivienne, S.: Designing energy-efficient convolutional neural networks using energy-aware pruning. In: Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6071–6079 (2016)
11. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural networks. In: Proceedings of Advances in Neural Information Processing Systems (NIPS), pp. 1135–1143 (2015)
12. Figurnov, M., Vetrov, D., Kohl, P.: PerforatedCNNs: acceleration through elimination of redundant convolutions, https://arxiv.org/pdf/1504.08362 (2015). Last Revised 16 Oct 2016
13. Bossard, L., Guillaumin, M., Van Gool, L.: Food-101—mining discriminative components with random forests. In: Fleet D., Pajdla T., Schiele B., Tuytelaars T. (eds.) Computer Vision—ECCV 2014. ECCV 2014. Lecture Notes in Computer Science, vol. 8694. Springer, Berlin (2014)
14. Forrest, N.I., Song, H., et al.: SqueezeNet: AlexNet-level accuracy with 50$\times$fewer parameters and $<$1 MB model size. https://arxiv.org/abs/1602.07360 (2016). Last Revised 4 Nov 2016
15. Qualcomm Neural Processing SDK for AI (https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk). Accessed 2019
16. Irida Labs S.A. (https://www.iridalabs.gr). Accessed 2020