Embedded Deep Neural Network Processing
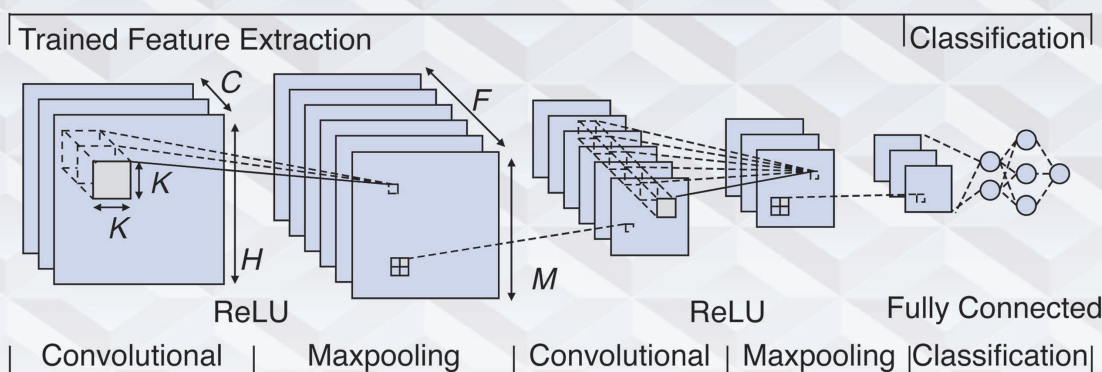
Algorithmic and processor techniques bring deep learning to IoT and edge devices

Trained Feature Extraction — Classification
Convolutional | Maxpooling | Convolutional | Maxpooling | Classification
ReLU — ReLU — Fully Connected

Marian Verhelst and Bert Moons

Deep learning has recently become im-mensely popular for image recognition, as well as for other recognition and pattern matching tasks in, e.g., speech processing, natural language processing, and so forth. The online evaluation of deep neural networks, however, comes with significant computational complexity, making it, until recently, feasible only on power-hungry server platforms in the cloud. In recent years, we see an emerging trend toward embedded processing of deep learning

networks in edge devices: mobiles, wearables, and Internet of Things (IoT) nodes. This would enable us to analyze data locally in real time, which is not only favorable in terms of latency but also mitigates privacy issues. Yet evaluating the powerful but large deep neural networks with power budgets in the milliwatt or even microwatt range requires a significant improvement in processing energy efficiency.

To enable such efficient evaluation of deep neural networks, optimizations at both the algorithmic and hardware level are required. This article surveys such tightly interwoven hardware-software processing techniques for energy efficiency

and shows how implementation-driven algorithmic innovations, together with customized yet flexible processing architectures, can be true game changers. To help readers fully understand the implementation challenges as well as opportunities for deep neural network algorithms, we start by briefly summarizing the basic concept of deep neural networks.

The Birth of Deep Learning
Deep learning [1] can be traced back to neural networks, which have been around for many decades and were already gaining popularity in the early 1960s. A neural network is a brain-inspired computing system,

typically trained through supervised learning, whereby a machine learns a generalized model from many training examples, enabling it to classify new items.

The trained classification model in such neural networks consists of several layers of neurons, wherein each neuron of one layer connects to each neuron of the next layer, as illustrated in Figure 1. The output of the network indicates the probability that a certain object class is observed at the network's input. In such a network, every individual neuron creates one output o, which is a weighted sum of its inputs i. For the nth neuron, of layer l, this can be formalized as

$$o_{ln} = \sigma\left(\sum_m w_{lmn} \cdot i_{lm} + b_{ln}\right). \quad (1)$$

The weights w_{lmn} and biases b_{ln} are the flexible parameters of the network that enable it to represent a particular desired input/output mapping for the targeted classification. They are trained with supervised training examples in an initial offline training phase, after which the network can classify new examples presented to its inputs, a process typically referred to as *inference*.

Such neural networks have been used for decades in several application domains. In a classical pattern-recognition pipeline [Figure 2(a)], features are generated from an input image by an application-specific feature extractor, hand-designed by an expert engineer. This preliminary feature extraction step was necessary because, at that time, one could use only small neural networks with a limited number of layers that did not have the modeling capacity required for complex feature extraction from raw data. Larger neural networks were impossible to train due to nonconvergence issues, lack of sufficiently large data sets, and insufficient compute power.

Yet, after a long winter for neural networks in the 1970s and 1980s, they regained momentum in the 1990s and again in the 2010s. The increasing availability of powerful compute servers and graphics processing units (GPUs), the abundance of digital data sources, and innovations in training mechanisms allowed training deeper and deeper networks, with many layers of neurons. This meant the start of a new era for classification, as it allowed training networks with enough
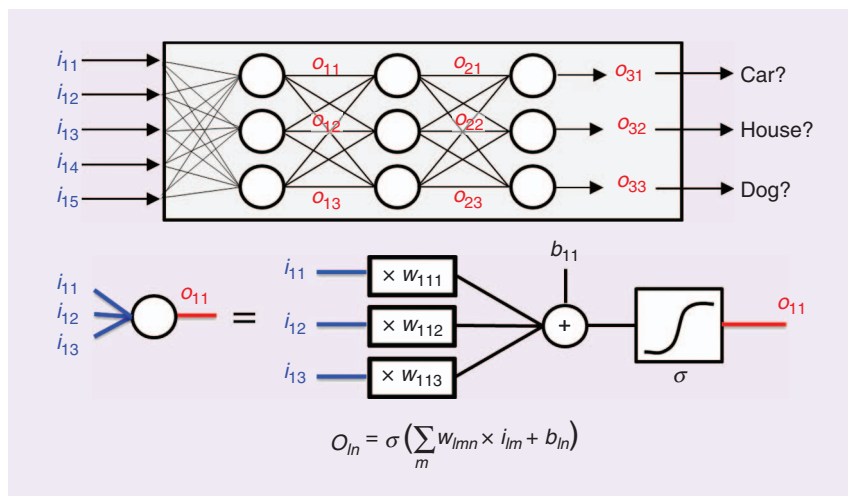

FIGURE 1: A traditional fully connected neural network is made up of layers of neurons. Every neuron makes a weighted sum of all its inputs, followed by a nonlinear transformation.
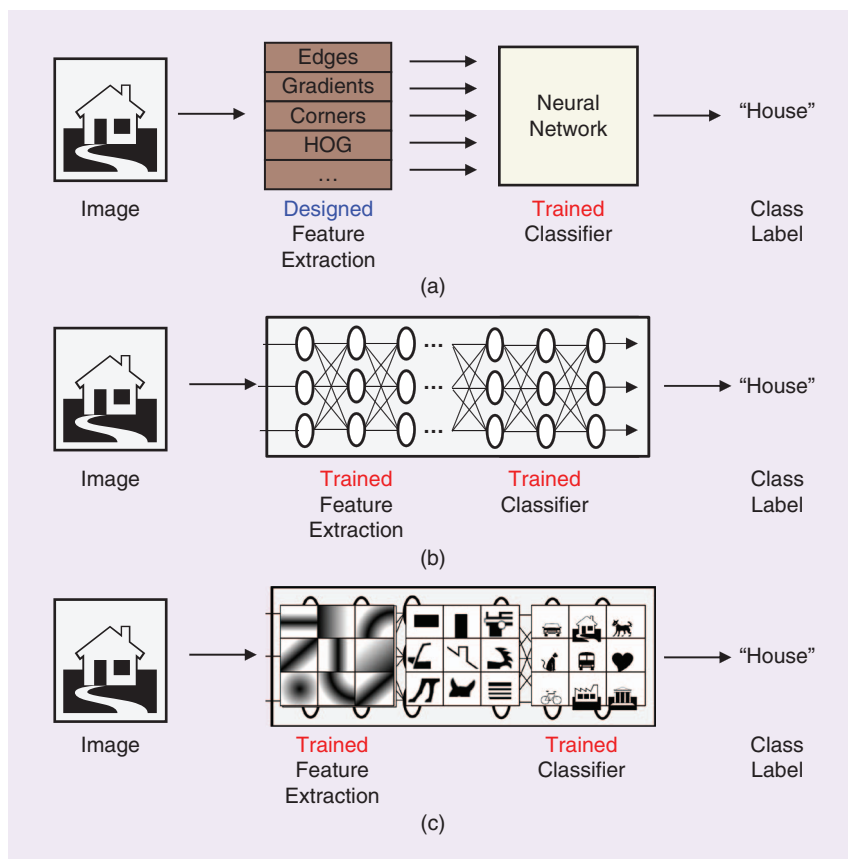

FIGURE 2: (a) Traditionally, machine learning classifiers were trained and applied on hand-crafted features. (b) The advent of deep learning allowed the network to learn and extract the optimal feature sets. (c) Such a network trains itself to extract very coarse, low-level features in its first layers, then finer, higher-level features in its intermediate layers, and, finally, targets full objects in the last layers. HOG: histogram of oriented gradients.

modeling capacity to operate directly on raw data. [Figure 2(b)]. Such "deep learning networks" thus fulfilled the role of both feature extractor and classifier.

A deeper network can automatically learn the best possible features during its training phase, instead of relying on features hand-crafted by humans. When inspecting trained networks, one can see that a deep neural network trains itself to extract very coarse, low-level features in its first layers and finer, higher-level features in its intermediate layers and then targets full objects in the last layers [Figure 2(c)].

A network's ability to learn the most optimal features significantly boosted the classification accuracy of such networks, resulting in their true breakthrough: deep learning was born. Over the last decade, deep learning has, as such, been able to move to deeper and deeper network architectures, enabling tremendous improvements in achievable classification accuracy, as illustrated by the results from the yearly ImageNet challenge (Figure 3) [2].

Deep Neural Network Topologies

Another crucial factor in the breakthrough of deep learning technology is the advent of new network topologies. Classical neural networks—which rely on so-called fully connected layers, with each neuron of one layer connected to each neuron of the next layer (Figure 1)—suffer from a very large number of training parameters. For a network with L layers of N neurons each, $L.(N^2 + N)$ parameters must be trained. Knowing that N can easily reach the order of a million (e.g., for images with a million pixels), this large parameter set becomes unpractical and untrainable.

For many tasks (mainly in image processing and computer vision), convolutional neural networks (CNNs) are more efficient. These CNNs, inspired by visual neuroscience, organize the data in every network layer as three-dimensional (3-D) tensors.

> *To enable efficient evaluation of deep neural networks, optimizations at both the algorithmic and hardware level are required.*

The first part of the network consists out of a sequence of convolutional layers and pooling layers, replacing the traditional fully connected layers. A convolutional layer transforms a 3-D input tensor O (of size $H \times H \times C$) into a 3-D output tensor I (of size $M \times M \times F$).

As illustrated in Figure 4, each element of the output tensor O does not need all elements of the input tensor I to be computed. Instead, it
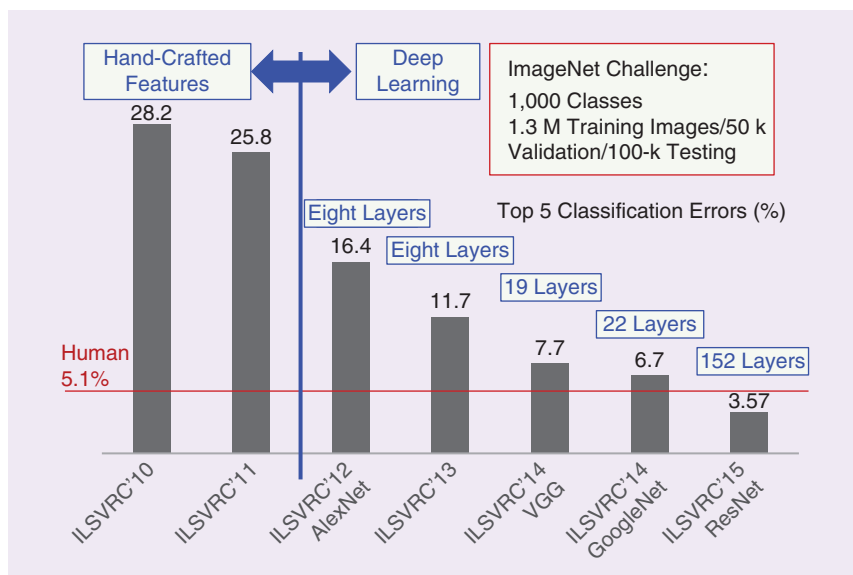

FIGURE 3: The classification results of the ImageNet challenge have seen enormous boosts in accuracy since the appearance of deep learning submissions. (Data from [2].) ILSVRC: ImageNet Large-Scale Visual Recognition Challenge; AlexNet: a CNN named for Alex Krizhevsky; VGG: a network from the Visual Geometry Group at Oxford University; ResNet: Residual Net.
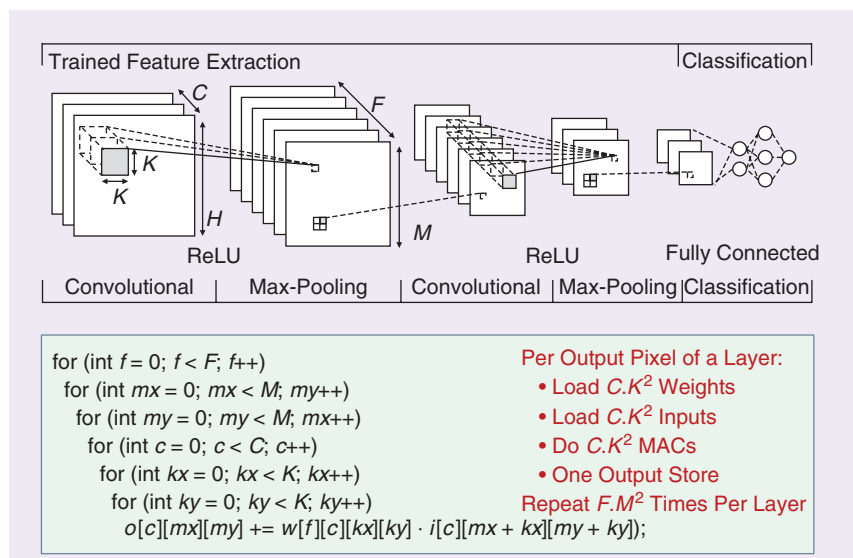


```
for (int f = 0; f < F; f++)
  for (int mx = 0; mx < M; my++)
    for (int my = 0; my < M; mx++)
      for (int c = 0; c < C; c++)
        for (int kx = 0; kx < K; kx++)
          for (int ky = 0; ky < K; ky++)
            o[c][mx][my] += w[f][c][kx][ky] · i[c][mx + kx][my + ky]);
```

Per Output Pixel of a Layer:
- Load $C.K^2$ Weights
- Load $C.K^2$ Inputs
- Do $C.K^2$ MACs
- One Output Store
Repeat $F.M^2$ Times Per Layer

FIGURE 4: The topology and pseudocode of one layer of a typical CNN. The psuedocode is for one layer of the network. MACs: multiply accumulation.

is connected only locally to a patch of the input tensor of size ($K \times K \times C$) through a trainable 3-D kernel W (of size $K \times K \times C$) and a bias B. A formal mathematical description to compute the outputs of a convolution layer, l, is given as

$$O_{lfxy} = \sum_{c=0}^{C} \sum_{i=0}^{K} \sum_{j=0}^{K} I_{lc(x+i)(y+j)} . W_{lfcij} + B_{lf}.$$

The result of the local sum computed in this filter bank is then passed through a nonlinearity layer, typically a rectified linear unit (ReLU), using the nonlinear activation function $f(u) = \max(0, u)$. This output can finally be processed by a max-pooling layer, which outputs only the maximum of a local patch (typically 2×2 or 3×3) of output units to the next layer. This thereby reduces the dimension of the feature representation and creates invariance to small shifts and distortions in the inputs. A modern CNN consists of tens [3] to hundreds [4] of such alternating convolutional and max-pooling layers, typically followed by one to three classification layers, implemented using the traditional fully connected neurons (Figure 4).

It is important to note that the same convolution kernel W and bias B are used to compute all (M X M) outputs of one slice in the output tensor. As such, every layer of the network needs only $F x (K \times K \times C + 1)$ parameters. With K typically ranging between one and seven and F and C on the order of tens or hundreds, this method allows the creation of very large networks while keeping the number of trainable parameters under control—all of which gave deep learning its significant boost.

The majority of recent state-of-the-art deep learning networks rely on such CNNs. The optimal network architecture, characterized by the number of cascading stages and the values of model parameters F, H, C, K, and M, varies for each specific application. Over the last few years, various alterations have been proposed to this standard topology, such as, e.g., introducing feed-through connections in ResNets [4], concatenating very small convolutions in inception networks [5], stacking depthwise and pointwise convolutions in Xception networks [6], extracting full-image dense multiscale features using DenseNets [7], or recurrent connections in RNNs or long short-term memories [8]. These, however, lie beyond the scope of this tutorial.

Challenges for Embedded Deep Inference

Both the training of a deep network and its own inferences to perform new classifications are now typically executed on power-hungry servers and GPUs [Figure 5(a)]. There is, however, a strong demand to move the inference step, in particular, out of the cloud and into mobiles and wearables to improve latency and privacy issues [Figure 5(b)]. However, current devices lack the capabilities to enable deep inferences for real-life applications.

Recent neural networks for image or speech processing easily require more than 100 giga-operations (GOP)/s to 1 tera-operations (TOP)/s, as well as the ability to fetch millions of network parameters (kernel weights and biases) per network evaluation. The energy consumed in these numerous operations and data fetches is the main bottleneck for embedded
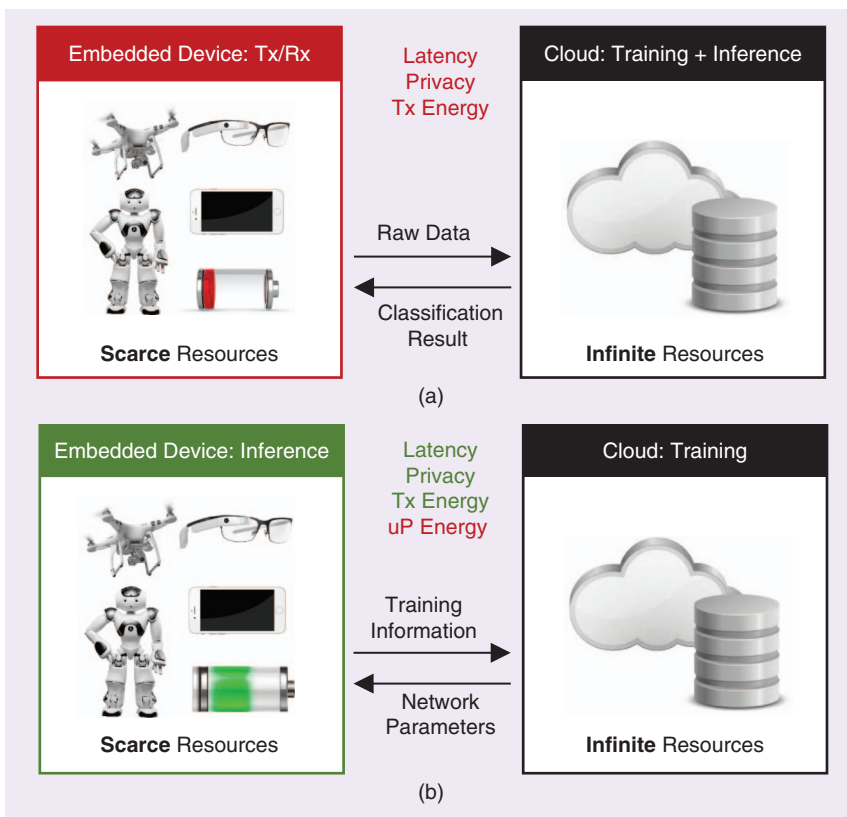

FIGURE 5: Concerns regarding user privacy, recognition latency, and energy wasted on raw data transmission push deep learning inferences from (a) the cloud to (b) the embedded device. Tx/Rx: transmitter/receiver; uP = microprocessor.

inference in energy-scarce milliwatt or microwatt devices. Currently, microcontrollers and embedded GPUs are limited to efficiencies of a few tens to hundreds of GOP/W, while embedded inference will only be fully enabled with efficiencies well beyond 1 TOP/W. Overcoming this bottleneck is possible yet requires a tight interplay between algorithmic optimization (modifying the network topology) and hardware optimization (modifying the processing architectures).

The following section elaborates on the most promising optimizations currently being explored toward energy-efficient, embedded deep inference. The focus here is on the energy-efficient execution of convolutional layers, which form the bulk of the workload during inference. However, several techniques can also be applied to fully connected layers.

Algorithmic and Architectural Techniques for Energy Efficiency

GPUs and central processing units (CPUs) are extremely flexible, general-purpose machines. While this makes them widely deployable and easy to use and program, it also limits their efficiency because they cannot exploit several computational aspects of deep inference networks, resulting in both a memory bottleneck and a computational bottleneck. More specifically, deep inference networks have three typical characteristics that can be exploited—or further enhanced—to improve execution energy efficiency:

1) Deep learning networks exhibit a very particular data flow with a large amount of potential parallelism and data reuse. This can, moreover, be manipulated during network training by playing with the F, H, C, K, and M parameters of the network.

2) Deep learning networks prove to be quite robust to approximations or fault introductions. This is exploited in various reduced-precision hardware implementations. Also, this characteristic can

be manipulated when training the network, allowing it to find the best tradeoff between a low complexity and a robust network.

3) Deep learning networks demonstrate large sparsity. Many parameters become very small, even equal to zero, after network training. Also, many data values propagated with the network during evaluation become zero. This can be exploited to reduce operations and memory fetches in hardware yet can also be stimulated further with innovative training techniques.

We will show how, for each of these three aspects, hardware can benefit from the network's characteristics but also how, during the algorithmic training phase of the network, it is possible to additionally optimize the particular characteristic to reach even greater efficiency gains. As such, it is clear that the hardware and algorithmic level need to closely cooperate not only to exploit but also to enhance the network's characteristics toward the most efficient hardware-software realization. All of the

techniques highlighted in this article are summarized in Figure 6.

Enhancing and Exploiting Network Structure

In many application areas, designers have improved the energy efficiency of embedded network evaluation by moving away from general-purpose processors and developing customized hardware accelerators. Such accelerators can exploit the known data flows within the algorithm to 1) enhance the parallel execution of the algorithm as well as 2) minimize the number of data movements (Figure 7). Descriptions of several application-specific integrated circuits targeting the efficient execution of convolutional and fully connected layers have recently been published.

All solutions exhibit a very large degree of parallelization, far beyond CPU parallelism. This easily demonstrates itself in a data path containing a few hundred to thousands of multiply accumulators (MACs), with Google's recent tensor processing unit as an extreme example (64,000 MACs) [9].
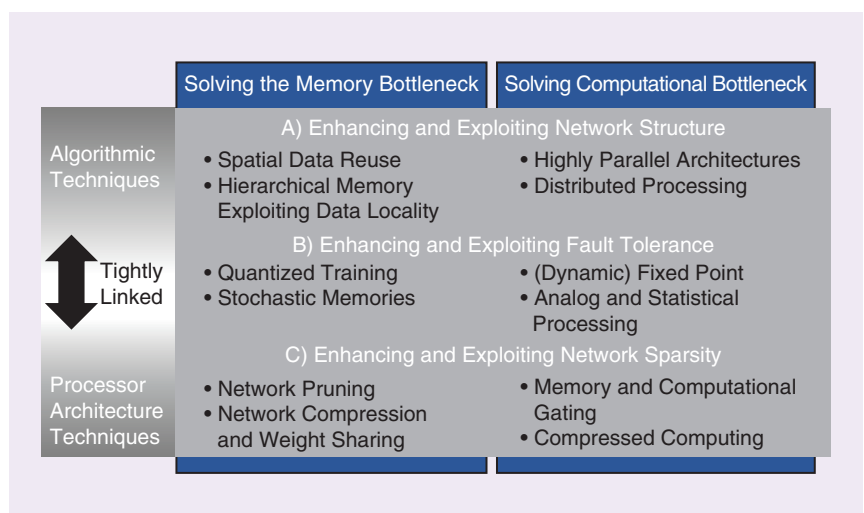
> **CNNs, inspired by visual neuroscience, organize the data in every network layer as 3-D tensors.**

FIGURE 6: An overview of the algorithmic and processor architecture techniques discussed to increase efficiency and enable the inference of deep neural networks in embedded devices.

Providing data to all these functional units in parallel would be nearly impossible if the temporal and spatial locality of the data was not exploited. Indeed, many computations within one network layer share common inputs. More specifically, as highlighted in the pseudocode shown in Figure 4, every weight parameter is reused approximately M^2 times across multiple convolutions of the same slice in the output tensor, and every input data point is reused across F different slices of the output tensor. Moreover, the intermediate accumulation results o have to be accumulated $C.K^2$ times. This can, in a custom accelerator, be exploited in several ways to further boost efficiencies beyond the highly parallel, yet not data-flow-optimized, GPUs.

Data reuse can be exploited by reusing the same data across multiple parallel execution units or, equivalently, across multiple time steps on the same execution unit. In this topology, three extreme cases can be distinguished, as shown in Figure 8.

The first multiplies the same input data value with several weights of a layer's different output channels. This is also called *weight parallel* or *input stationary*. In this implementation, every input will ideally be loaded into the system only once. This, however, has negative repercussions on the weight memory bandwidth, as the weights must be reloaded frequently (every time a new input is applied). Moreover, the accumulation of the output o cannot be performed across different clock cycles,

requiring intermediate accumulation results o to be pushed into memory and refetched later, strongly impacting the input/output memory bandwidth. A similar scheme fetches every weight once and multiplies it with many input values. This "weight stationarity" or "input parallelism" improves the weight memory bandwidth, yet at the expense of the input memory bandwidth. Finally, the output stationary scheme reloads new weights and inputs every single clock cycle and yet is able to accumulate the intermediate results locally within the MAC unit across different clock cycles, to the benefit of the output memory bandwidth.

All these optimizations can be seen as a reshuffling of the nested loops in the pseudocode of Figure 4. Of course, in practice, most realizations implement a hybrid form of the three presented extreme cases. Examples include [23] and [24], where a two-dimensional (2-D) data path multiplies every input with several weights, while every weight is also multiplied with several inputs, and [10], where the input and output stationarities are optimized to minimize the chip input/output bandwidth. Which parallelization scheme is optimal depends strongly on the network's dimensions; the parameters F, H, C, K, and M, which allow cooptimization of the hardware; and
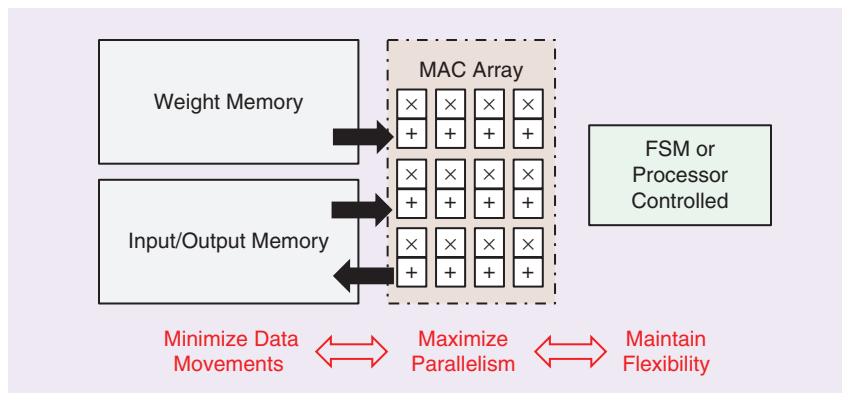

FIGURE 7: Custom deep neural network processors gain efficiency by minimizing data movements and maximizing parallelism. Still, it is crucial not to lose all flexibility in mapping a wide variety of networks. FSM: final state machine.
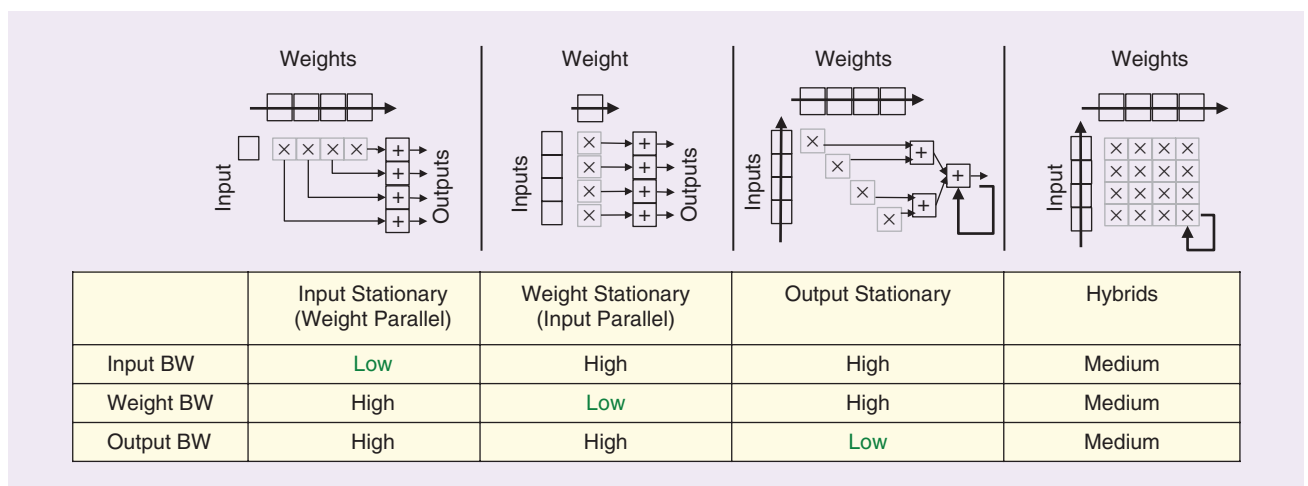

	Input Stationary (Weight Parallel)	Weight Stationary (Input Parallel)	Output Stationary	Hybrids
Input BW	Low	High	High	Medium
Weight BW	High	Low	High	Medium
Output BW	High	High	Low	Medium

FIGURE 8: Different architectural topologies allow data reuse to be maximized, reusing either inputs, weights, intermediate results, or a combination of the three. BW: bandwidth

the network itself. A more elaborate overview of the different parallelization schemes can be found in [11] and [12], along with an assessment of their merits.

A complementary way to reduce the energy burden of continuous data fetches is not to minimize the number of data fetches but rather to reduce the energy cost of every data fetch by exploiting temporal data locality. Most realistic deep networks require so much weight and input/output memory (megabytes to gigabytes) that it is impossible to fit them in on a chip memory, thus requiring fetches from energy-costly external dynamic random-access memory (DRAM). Similar to traditional processors, this can, however, be mitigated by a memory hierarchy having one or more levels of on-chip static RAM (SRAM) or register files. Frequently accessed data can, as such, be stored locally to reduce its fetching cost (Figure 9).

An important difference with general-purpose solutions, however, is that the sizes of the memories in the hierarchy can be optimized toward the network's structure, e.g., foreseeing a local memory capable of caching exactly one weight tensor, or one of the tensor [11]. Even more importantly, the networks can be trained with the processor's memory hierarchy in mind. As such, networks have, e.g., been explicitly trained to completely fit in on-chip memory. This optimization is, of course, highly interwoven with the parallelization scheme. By jointly optimizing these, one can adjust the degree of parallelization to the memory hierarchy and minimize the product of the number of memory accesses with the cost of every memory access [13].

Distributed and systolic processing can be seen as an extreme type of such hierarchical memories. In the systolic processing concept, a 2-D array of functional units processes data locally and passes inputs and intermediate results from unit to unit instead of to/from global memory. These functional units are each equipped with a very small SRAM (as

In the systolic processing concept, a 2-D array of functional units processes data locally and passes inputs and intermediate results from unit to unit instead of to/from global memory.

in [14]) or even just registers (as in [9]) to store data locally and maximize data reuse within the array. Processing happens as a systolic wavefront through the array, wherein weight coefficients can be kept stationary in the functional units, input data are shifted in one direction through the array, and output data accumulate in the orthogonal direction. This allows the performance of a very large number of computations for convolution or matrix multiplication in parallel by keeping all systolic elements busy without burdening the memory bandwidth. Interested readers are pointed to [15] and [9] for more details.

Such systolic operation opens the door to in-memory computing, where the computation is integrated inside the memory array. While this is also pursued in traditional memory architectures, the results look especially promising for emerging nonvolatile memory arrays. For example, in resistive memory technologies, a multiplication can be implemented by exploiting the memory cell's conductance as the kernel weight, while accumulating current from different elements to implement the convolution's accumulation operation [16]. However, this technology currently still suffers from large variability, limiting applications to very low-resolution operations with very limited kernel and network sizes.

While all the aforementioned techniques can dramatically boost the system's throughput and energy efficiency, it is important to keep an eye on their impact on the design's programmability and flexibility. Especially in the fast-paced area of deep learning, it is of the utmost importance to maintain sufficient flexibility toward alternative network dimensions and novel network topologies. Most accelerators, however, succeed in this by enabling the acceleration of matrix multiplications (for the fully connected layers) and convolutions (for the convolution layers) of any size, yet with maximal efficiency for a subset of sizes.

Enhancing and Exploiting Fault Tolerance
A second important aspect of deep neural networks that can be exploited in custom processor designs is their fault tolerance. Many studies observe the robustness of CNNs and other networks to perturbations on their weight parameters and intermediate computational results [17], [18]. This can be exploited both at the hardware as well as the algorithmic level in several ways.
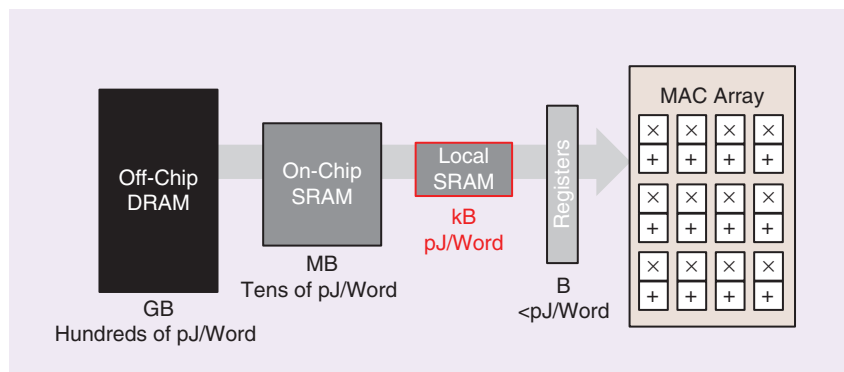

FIGURE 9: A well-designed memory hierarchy avoids drawing all weights and input data from the costly DRAM interface and stores frequently accessed data locally. pJ: picojoule.

A straightforward way to benefit from the network's fault tolerance is to perform the computations at reduced computational accuracy with limited recognition loss. Typical benchmarks can be run at a 1–9-b fixed point rather than a 32-b floating point at lower than 1% accuracy loss [18]. This is possible by quantizing all weights of a floating-point-trained network before execution. Improved results can be obtained when introducing quantization during the training step itself [19], [38], resulting in smaller or lower-precision networks for the same application accuracy. As an extreme example, networks have been specifically trained to operate with only 1-b representations of weights alone [20] as well as with both weights and activations [20], [21] wherein all multiplications can be replaced by efficient XNOR operations [22]. In [20], a binary-weight version of ImageNet is only 2.9% less accurate (in top-1 accuracy) than the full-precision AlexNet [3].

This observation can lead to major energy savings, as current CPU and GPU architectures operate using 32–16-b floating-point number formats. Reducing precision from 32-b floating point to low precision not only reduces computational energy but also minimizes the storage and data-fetching cost needed for network weights and intermediate results. Moreover, for very low bit widths, this even allows the replacement of multipliers that have several data values with a common weight factor via preloaded lookup tables [10]. As a result, all custom CNN accelerators operate in fixed point. While most processors operate at constant 16-, 12-, or 8-b word lengths, some recent implementations support variable word-length computations, wherein the processor can change the used computational precision from operation to operation [23], [10], [24]. This accommodates for the observation that the optimal word length for a deep network strongly varies from application to application and is even shown to differ across various layers of a single deep network [18] [Figure 10(a)].

Energy-efficient variable-resolution processors have been realized using a technique termed *dynamic voltage-accuracy-frequency scaling* [25] to jointly reduce the switching activity, supply voltage, and parallelization scheme when computational resolution drops [Figure 10(c)]. This results in a scaling of the system's energy consumption, which is super-linear with the computational resolution [Figure 10(b)], thus allowing every network layer to run at its own minimal energy point. Reduced

bit-width implementations all exploit the deep network's tolerance to faults in a deterministic way.

Another school of thought targets energy savings through tolerating nondeterministic statistical errors. This can be accomplished by executing the convolutional kernels in the noisy analog domain [26]. Alternatively, in the digital domain, stochastic fault tolerance can be exploited by operating the circuits [27] and/or memory [28] in the energy-efficient near-threshold regions. In this region, circuit delays as well as memory failures suffer from large variation. Yet the networks can tolerate such stochastic behavior up to a certain limit. Such circuits are combined with circuit monitors that constantly assess and control the circuit's fault rate [28].

Finally, the operational circumstances can strongly influence the network's tolerance to approximations. In a given classification application, the quality of the inputs might change dynamically, or some classes might be easier to observe than others. If one tries to train one common network that performs acceptably under all possible circumstances and classes, a large, complex, energy-hungry network topology would be needed. Recent work, however, promotes the training of hierarchical or staged
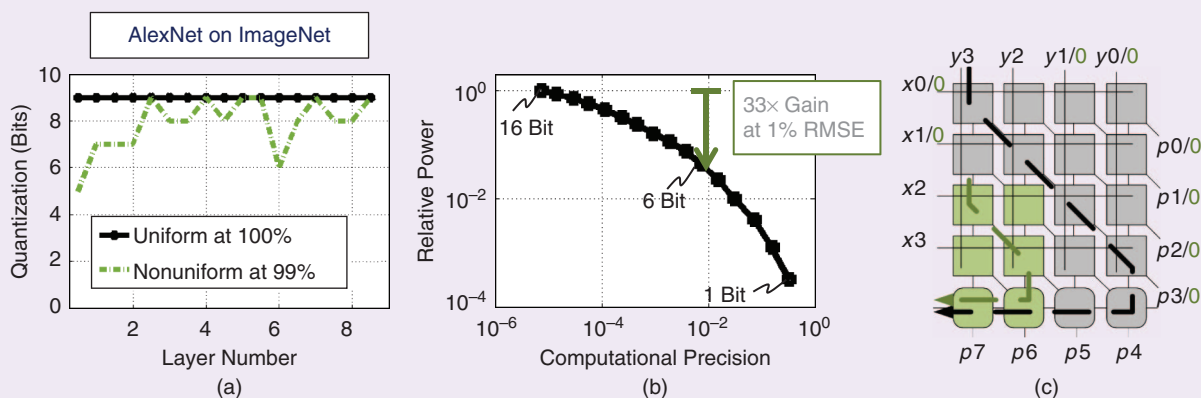

FIGURE 10: (a) When quantizing all weight and data values in a floating point AlexNet uniformly, the network can run at 9-b precision. Lower precision can be achieved without significant classification accuracy loss by running every layer at its own optimal precision. This allows (b) saving power in the function of computational precision and (c) building multipliers whose energy consumption scales drastically with computational precision, through reduced activity factor and critical path length.

networks [29] that perform classifications in several optional stages. At each stage, only a few layers of the network are executed, after which a classification layer tries to guess the class from the current outputs. Additional network layers and classifiers are run only if the obtained probabilities are not outspoken enough, until a classification with distinct probabilities is obtained. Such dynamic evaluations can be performed on any hardware platform but, again, benefit significantly from implementation-aware training techniques or topology-optimized implementations. Inference on the ImageNet data set [29] required up to 2.6 times fewer operations than state-of-the-art networks at equivalent accuracy.

Enhancing and Exploiting Sparsity

Deep neural networks exhibit extreme sparsity, i.e., many of the weight values, as well as intermediate data values, are zero. Figure 11(a) shows the sparsity of an AlexNet in function of the used fixed-point word length within the network. As can be seen, even for large word lengths, more than 70% of the activations are zero. At reduced bit-width computations, also many weight values are quantized to zero. This opens up many opportunities.

On the hardware side, this can be exploited by preventing any MAC with a zero-valued input [see Figure 11(b)], by not even fetching zero-valued data values from memory, and by strongly

> ## An important difference with general-purpose solutions, however, is that the sizes of the memories in the hierarchy can be optimized toward the network's structure.

compressing the on/off chip data stream using, e.g., Huffman or other types of encoding. Several hardware implementations exploit these CNN characteristics. The authors of [24] and [11] skip all unnecessary sparse operations by gating the inputs to their arithmetic units if the input data is zero, as a multiply-accumulate with zero does not change the internal accumulation result. Both implementations also compress off-chip data streams, either through run-length encoding [14] or through a simplified Huffman scheme [23]. The architectures presented in [30] and [31], on the other hand, allow speeding up sparse network evaluation s by only scheduling non-zero operations for execution, improving computational throughput up to 1.52 and 5.2 times, respectively.

More powerful opportunities arise, again, when the hardware and algorithmic plane are jointly involved. Deep network training algorithms can be modified to enhance the network's sparsity by iteratively pruning the smallest weight values (quantizing them to zero) and retraining the network [32]. Going one step further, energy-aware pruning techniques even take the energy consumption

model of the hardware into account and start pruning the layers that consume the most energy, to maximize pruning efficacy [33]. This easily allows the pruning of 70–90% of the weights and saves up to 70% of energy consumption.

Interestingly enough, networks have more compression capabilities beyond simply that of pruning low-valued weights. After pruning and quantizing a network, it turns out that the resulting weight values are highly clustered. This allows, e.g., the clustering of 8-b weights in only 16 (2^4) different weight clusters, each of which can share a common weight value expressed by a 4-b label. For every weight value, only the 4-b labels are stored, and these are expanded online to their original 8-b value using a small embedded lookup table.

Recent work has shown that the combination of pruning, weight sharing, and Huffman compression compresses state-of-the-art networks by 50 times in memory size (deep compression [32]). Traditional accelerators can benefit from such compression but only in terms of a reduction in memory size and the amount of memory accessed. To execute convolutional operations, they must still
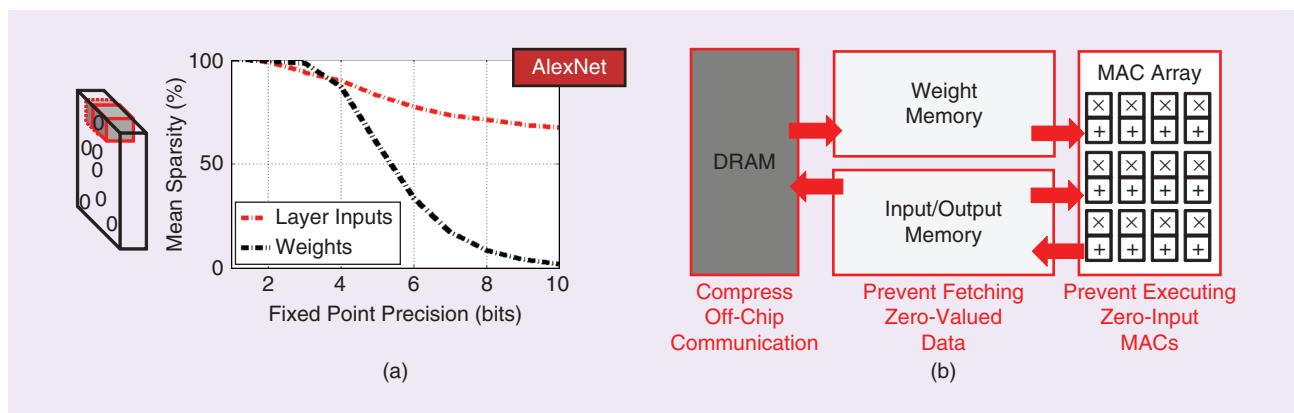

FIGURE 11: (a) The sparsity of input and weight values of a typical network in function of computational precision at which the network is evaluated. (b) This sparsity allows energy to be saved in the processor's input/output interface, on-chip memories, and data path.

decompress the data and, at best, remain idle during zero-valued operations. The efficient-inference engine [35], however, demonstrates that it is also possible and highly beneficial to operate directly on the compressed data by adapting the data path and memory interface to the compressed data format.

A network compression technique that does enable straightforward network execution in the complex domain without any hardware adaptation uses singular value decomposition (SVD) [36]. By performing SVD on a sparse weight matrix of a fully connected network layer, the matrix can be decomposed into two matrices, the rows and columns of which are ordered by the function of the most significant network parameters. By simply removing the nonsignificant sections of the matrix, one is left with a strongly compressed representation of the original network layer. The result can be executed on any regular neural network accelerator, as it is identical to the execution of two (much smaller) fully connected layers. While this method is more straightforward from a hardware point of view, it

offers only limited compression capabilities, ranging typically up to only five times compression [36].

Outlook

In this short tutorial, we have presented a selection of very promising hardware and algorithmic techniques from the rapidly expanding and growing field of deep learning. Each exploits and/or enhances the unique features of deep networks to improve the energy efficiency of their execution. Together, they have allowed the achievement of tremendous energy savings compared to traditional CPU- and GPU-based compute platforms. As can be seen in Figure 12 [37], this recent wave of innovations breaks the barrier for embedded deep inference in mobile devices. Implementations far surpassing the efficiencies of 1-TOP/W have recently been demonstrated, while computational throughput is boosted to several 100 GOP.

Still, challenges remain to effectively bring deep learning to IoT and edge devices. First, few (if any) complete end-to-end solutions have been demonstrated. Doing so involves

integrating the deep-inference chips in complete vision-processing pipelines mapping real-life applications. This requires not only an efficient execution of the inference kernel itself but also efficient image slicing, data transfer, and results interpretation.

A second interesting challenge lies in the learning process. So far, most chips focus on the inference part, where pretrained models are efficiently executed on-chip. In the future, however, the desire for more privacy and user customization will stimulate chips capable of executing the training phase as well. This, however, comes with new computational challenges and the need for a careful algorithm–architecture cooptimization.

It is, thus, very clear that, more than ever, the hardware and algorithmic layer must be optimized jointly, grasping the various cross-layer opportunities of deep neural networks. This is also apparent from the interest of many traditionally software-oriented companies (like Google, Amazon, and Microsoft) in the development of new proprietary hardware for deep learning.

This field is so vibrant that every single week new ideas pop up. Of course, space does not allow us to cover all of the exciting ideas going around in the embedded deep learning space at the moment. Yet we hope that we were able to spark readers' interest and stimulate further exploration of this lively field.
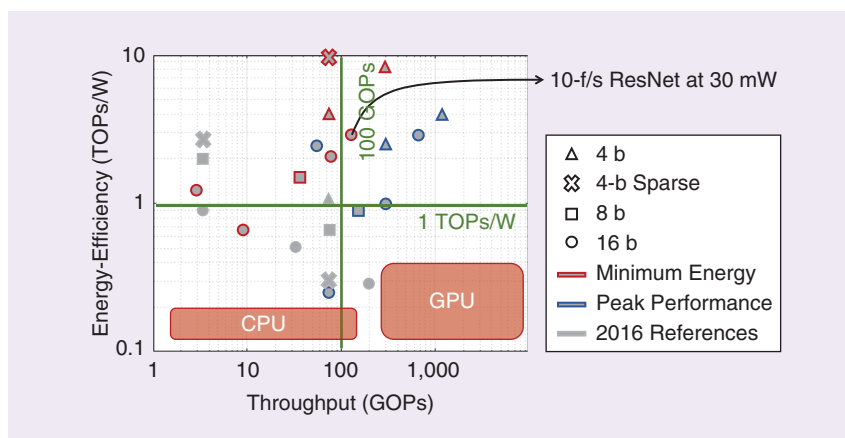

FIGURE 12: An overview of the reported performance of the deep neural network processors published at the International Solid-State Circuits Conference in 2016 and 2017. Performances beyond 100 GOP and 1 TOP/W will be a game changer for deep inference in embedded devices.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp 436–444 2015.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *Int. J.Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[3] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Conf. Neural Information Processing Systems*, 2012, pp. 1097–1105.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv Preprint*, arXiv:1512.03385, 2015.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, and A. Rabinovich, "Going deeper with convolutions," in *Proc.*

IEEE Conf. Computer Vision and Pattern Recognition, 2015, pp. 1–9.

[6] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *arXiv Preprint*, arXiv:1610.02357, 2016.

[7] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, "Densenet: Implementing efficient convnet descriptor pyramids," arXiv Preprint, arXiv:1404.1869, 2014.

[8] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, 2000.

[9] N. P. Jouppi, et al. "In-datacenter performance analysis of a tensor processing unit," *arXiv Preprint*, arXiv:1704.04760, 2017.

[10] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 240–241.

[11] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. IEEE Annu. Int. Symp. Computer Architecture*, 2016, pp. 367–379.

[12] M. Peemen, et al. "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Computer Design*, 2013, pp. 13–19.

[13] L. Cecconi, S. Smets, L. Benini, and M. Verhelst, "Optimal tiling strategy for memory bandwidth reduction for Cnns: Advanced concepts for intelligent vision systems," Ph.D. dissertation, Univ. Bologna 2017.

[14] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2016, pp. 262–263.

[15] H. T. Kung, "Systolic algorithms for the CMU WARP processor," Research Showcase @ CMU, 1984.

[16] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramanian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. 43rd Int. Symp. Computer Architecture*, 2016, pp.14–26.

[17] P Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," in *Proc. Workshop Contribution to Int. Conf. Learning Representations*, 2016.

[18] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient ConvNets through approximate computing," in *Proc. IEEE Winter Conf. Applications Computer Vision*, 2016, pp. 1–8.

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," arXiv preprint, arXiv:1609.07061, 2016.

[20] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Proc. European Conf. Computer Vision*, 2016, pp. 525–542.

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural networks in advances" in *Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Assoc., Inc. 2016, pp. 4107–4115.

[22] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights," in *Proc. IEEE VLSI Computer Society Annu. Symp.*, July 2016, pp. 236–241.

[23] B. Moons and M. Verhelst, "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets," in *Proc. IEEE Symp. VLSI Circuits*, 2016, pp. 1–2.

[24] B. Moons, et al. "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 246–257.

[25] B. Moons, R. Uyttterhoeven, W. Dehaene, and M. Verhelst, "DVAFS: Trading computational accuracy for energy through dynamic-voltage-accuracy-frequency-scaling," in *Proc. Conf. Design, Automation and Test in Europe*, Lausanne, 2017, pp. 488–493.

[26] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, and D. Fick, "Analog in-memory subthreshold deep neural network accelerator," in *Proc. IEEE Custom Integrated Circuits Conf.*, Austin, TX, 2017, pp. 1–4.

[27] Y. Lin, S. Zhang, and N. R. Shanbhag. "Variation-tolerant architectures for convolutional neural networks in the near threshold voltage regime," in *Proc. IEEE Int. Workshop Signal Processing Systems*, 2016, pp. 17–22.

[28] P. Whatmough, S. Kyu Lee, H. Lee, S. Rama, D. Brooks, and G.-Y. Wei, "A 28nm SoC with a 1.2GHz 568nJ/pred sparse deep neural network engine with >0.1 timing error rate tolerance for IoT applications," in Proc. *IEEE Int. Solid-State Circuits Conf.*, 2017, pp. 242–243.

[29] G. Huang, et al. "Multi-scale dense convolutional networks for efficient prediction," *arXiv Preprint*, arXiv:1703.09844, 2017.

[30] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Computer Architecture*, June 2016, pp. 1–13.

[31] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," in *Proc. IEEE Design, Automation & Test in Europe Conf. & Exhibition*, 2017, pp. 1462–1467.

[32] S. Han, J. Pool, J. Tran, and W. Dally. "Learning both weights and connections for efficient neural network," in *Proc. Advances in Neural Information Processing Systems*, 2015, pp. 1135–1143.

[33] V. Sze, T.-J. Yang, and Y.-H. Chen, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. Conf. Computer Vision and Pattern Recognition*, Honolulu, Hawaii, July 21–26, 2017, pp. 5687–5695.

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

[35] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *arXiv Preprint*, arXiv:1602.01528, 2016.

[36] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition," in *Proc. Interspeech Conf.*, 2013, pp. 2365–2369.

[37] M. Verhelst. (2017). Deep learning processor survey. [Online]. Available: http://www.esat.kuleuven.be/~mverhels/DLIC-survey.html

[38] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," arXiv preprint, arXiv:1606.06160.

About the Authors

Marian Verhelst (marian.verhelst@kuleuven.be) has been an assistant professor at the Micro-Electronics and Sensors Laboratories of the Electrical Engineering Department at KU Leuven, Belgium, since 2012. Her research focuses on self-adaptive circuits and systems, embedded machine learning, and low-power sensing and processing for the Internet of Things. She received a Ph.D. degree from KU Leuven (cum ultima laude) in 2008. She was a visiting scholar at the Berkeley Wireless Research Center of the University of California, Berkeley, in 2005. From 2008 to 2011, she worked in the Radio Integration Research Lab of Intel Laboratories, Hillsboro, Oregon. She is an IEEE Solid-State Circuits Society Distinguished Lecturer and a member of the Young Academy of Belgium and has published over 100 papers in conferences and journals. She is a member of the International Solid-State Circuits Conference (ISSCC) Technical Program Committee and the Design, Automation, and Test in Europe (DATE) and ISSCC Executive Committees. She was associate editor for *IEEE Transactions on Circuits and Systems II* and currently serves in the same capacity for *IEEE Journal of Solid-State Circuits*.

Bert Moons received his B.S. and M.S. degrees in electrical engineering from KU Leuven, Belgium, in 2011 and 2013, respectively. In 2013, he joined the Micro-Electronics and Sensors Laboratories of KU Leuven as a research assistant, funded through an individual grant from the Research Foundation of Flanders. In 2016, he was a visiting research student at Stanford University, California, in the Murmann Mixed-Signal Group. Currently, he is working toward the Ph.D. degree on energy-scalable and run-time adaptable digital circuits for embedded deep learning applications. *SSC*