Received October 23, 2020, accepted November 8, 2020, date of publication November 24, 2020, date of current version December 30, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3039858

Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead

MAURIZIO CAPRA[1], (Graduate Student Member, IEEE), BEATRICE BUSSOLINO[1], (Graduate Student Member, IEEE), ALBERTO MARCHISIO[2], (Graduate Student Member, IEEE), GUIDO MASERA[1], (Senior Member, IEEE), MAURIZIO MARTINA[1], (Senior Member, IEEE), AND MUHAMMAD SHAFIQUE[3], (Senior Member, IEEE)

[1]Department of Electrical, Electronics, and Telecommunication Engineering, Politecnico di Torino, 10129 Torino, Italy
[2]Institute of Computer Engineering, Technische Universität Wien (TU Wien), 1040 Vienna, Austria
[3]Division of Engineering, New York University, Abu Dhabi, United Arab Emirates

Corresponding author: Maurizio Capra (maurizio.capra@polito.it)

This work was supported in part by the Ph.D. College Resilient Embedded Systems which is run jointly by TU Wien's Faculty of Informatics and FH-Technikum Wien.

ABSTRACT Currently, Machine Learning (ML) is becoming ubiquitous in everyday life. Deep Learning (DL) is already present in many applications ranging from computer vision for medicine to autonomous driving of modern cars as well as other sectors in security, healthcare, and finance. However, to achieve impressive performance, these algorithms employ very deep networks, requiring a significant computational power, both during the training and inference time. A single inference of a DL model may require billions of multiply-and-accumulated operations, making the DL extremely compute- and energy-hungry. In a scenario where several sophisticated algorithms need to be executed with limited energy and low latency, the need for cost-effective hardware platforms capable of implementing energy-efficient DL execution arises. This paper first introduces the key properties of two brain-inspired models like Deep Neural Network (DNN), and Spiking Neural Network (SNN), and then analyzes techniques to produce efficient and high-performance designs. This work summarizes and compares the works for four leading platforms for the execution of algorithms such as CPU, GPU, FPGA and ASIC describing the main solutions of the state-of-the-art, giving much prominence to the last two solutions since they offer greater design flexibility and bear the potential of high energy-efficiency, especially for the inference process. In addition to hardware solutions, this paper discusses some of the important security issues that these DNN and SNN models may have during their execution, and offers a comprehensive section on benchmarking, explaining how to assess the quality of different networks and hardware systems designed for them.

INDEX TERMS Machine learning, ML, artificial intelligence, AI, deep learning, deep neural networks, DNNs, convolutional neural networks, CNNs, capsule networks, spiking neural networks, VLSI, computer architecture, hardware accelerator, adversarial attacks, data flow, optimization, efficiency, performance, power consumption, energy, area, latency.

I. INTRODUCTION

Artificial intelligence (AI) has become a fundamental pillar in many applications and systems in recent years. It is transforming the way we interact with technology, to the point that, very often, we use it without even realizing it. Many techniques fall under the domain of AI, while one in particular raised among all, the Machine Learning (ML). In the last two decades, ML has been extensively employed in various application domains, thanks to the wide range of flexible and easy to learn statistical patterns. ML further consists of several subtopics, as shown in Figure 1. The most popular ones are the

The associate editor coordinating the review of this manuscript and approving it for publication was Shiping Wen.

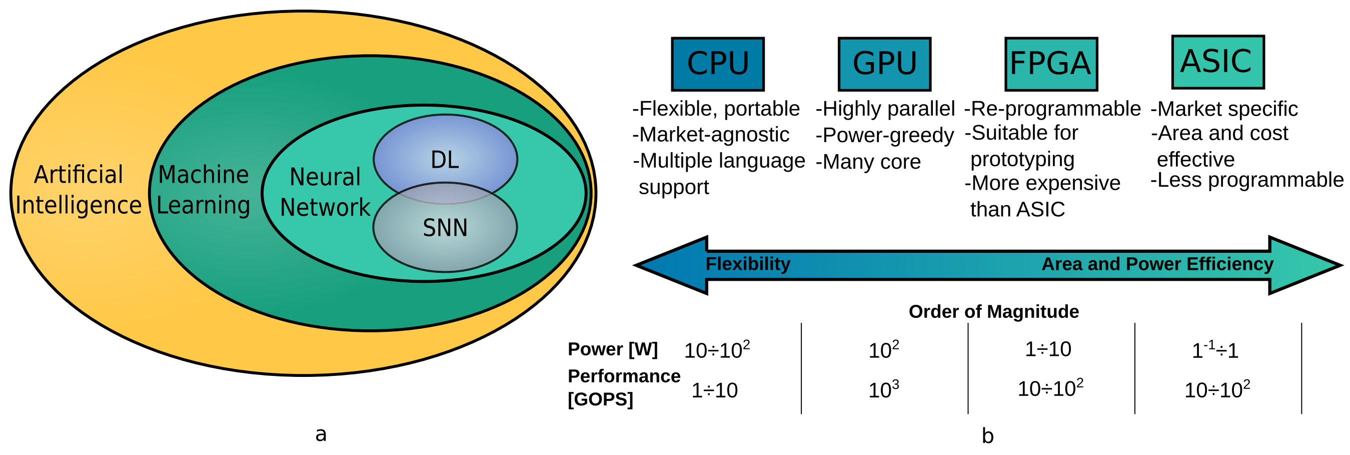


FIGURE 1. a) Artificial intelligence overview. b) Hardware platforms comparison [1].

brain-inspired models such as the Neural Networks (NNs), including the Spiking Neural Networks (SNNs) and the Deep Learning (DL) with Deep Neural Networks (DNNs).

DL shows superior accuracy, even claimed to exceed the human one in certain cases, e.g., in image classification and other problems of computer vision [2]. This is mainly enabled because of the two factors: (1) the computational power of the latest generation processors, and (2) the enormous amount of available data for training, from which DL can learn different patterns and can effectively derive certain predictions, using deeper and complex models. The larger the training dataset, the more and better the DL algorithms can learn and cover corner cases, achieving the performance never seen before. Since the training is a time-consuming task, effective hardware solutions are required to provide ready-to-use models within a reasonable time. This article mainly focuses on the hardware solutions related to those Deep Neural Networks (DNNs) that have captured much of the attention in the recent years, discussed in Section II. This article will also provide a brief overview of work on SNNs, which are becoming increasingly popular due to their similarities to the human brain and their energy-efficient computations. The applications that are already DL-based are numerous, and cover many key areas:

- **Computer Vision**: It is fundamental to extract meaningful features from video and pictures. Such tasks include object localization [3], image classification [4], and image segmentation [5]. Their use is valuable for controlling web traffic [6] or for example, video surveillance [7].
- **Business and Finance**: Financial techs deploy such models to forecast market behavior [8], including insurance [9] and lending [10].
- **Healthcare**: DL is widely used in cancer detection such as lung cancer [11], brain cancer [12], skin cancer [13], and many others are continuously rising. Moreover, there is also a wide applicability of DL techniques in the IoT-Healthcare use cases and Wearables, for instance, to derive short-term and long-term health predictions.
- **Robotics**: In robotics, DNNs served in a wide range of use cases like autonomous vehicles [14], humanoid robots [15], assistive robots [16], swarms [17], and drone control system [18].
- **Smart Energy Management**: DL can also be used to preserve valuable resources such as electricity. Indeed both managing [19] and forecasting [20] the required amount of energy consumption can lead to significant savings.

DNNs learn intelligent activities without the explicit handcrafted guidelines of experts. Although DNNs, particularly CNNs and RNNs, represent the state-of-the-art in a wide range of applications, their increasing complexity demands for powerful hardware. Indeed, both inference and training processes require tens of billions of multiply-and-accumulate (MAC) operations that make these models extremely compute-intensive. Moreover, for each MAC, at least two input elements must be fetched from memory. As a result, performing these algorithms with minimal latency entails an additional critical constraint over the memory bandwidth.

For the reasons stated above, in many cases CPUs are not enough, therefore GPUs are one of the most appealing alternative to execute such complex models. However, today's trend is driven by the Internet-of-Things (IoT) [21] applications that require more computation capability near the sensors. This process of moving resources towards the IoT nodes is also known as edge computing [22]. This has become possible for two main reasons. Firstly the cost per silicon area has fallen to such an extent that the production of large scale devices to embed in IoT nodes is not an impediment anymore. Secondly, by performing on-site operations, it is no longer necessary to transmit the data to a central server, thus distributing the computing capacity reduces both latency and the large amounts of energy required for transmission, as well as preserving the privacy of data of edge nodes. The mesh of these nodes is subjected to strict power constraints, indeed, many of them are battery-powered or rely on energy harvesting systems [23]. Therefore, the integration of a high-end GPU into such a system is unfeasible since the required power would go far beyond the power envelope of the IoT-edge platforms.

In this scenario, DL algorithms need to be accelerated with alternative technologies such as low-power FPGAs, that are flexible and can be reprogrammed, or specialized accelerators in form of ASIC-IPs that are highly optimized and tailored for the application use case. This is also justified by the recent trend of integrated systems to move towards heterogeneous multicore systems (or heterogeneous multi-processor system on chip, MPSoCs) [24], which embed a mix of low-power general-purpose cores and specialized hardware accelerators. The flexibility of FPGA and ASIC designs (Figure 1b) opens up a whole series of possible hardware optimizations, analyzed in the following, that are required for energy-efficient acceleration of DL models. This work analyzes several hardware aspects that different platforms (CPU, GPU, FPGA, and ASIC) provide for the acceleration of DNN models with a comprehensive focus on dedicated accelerators. The latter, as explained before, gained much attention in recent years, thanks to their low-power and cost-effectiveness processing profile. Having a broad overview of the latest state-of-the-art concepts and methodologies can be very valuable for designers.

Table 1 lists the acronyms used in this paper for a better understanding.

Paper organization: this survey paper is organized systematically in different sections and sub-sections, as depicted in Figure 2. Section II describes the background of DNNs and SNNs, describing the evolution of networks over the years and providing examples of DNN architectures considered the milestones of the DL. Section III analyses different co-design techniques to translate and map an efficient dataflow onto the hardware. Section IV outlines the characteristics of the memory hierarchy, being this an extremely power-greedy element. Section V presents the security issues related to ML models, providing examples on how to handle them. Section VI identifies the most important DL frameworks besides the datasets and the essential metrics to characterize both models and hardware devices. Section VII provides some hints about the research trends and future directions of ML and DL. Section VIII provides a description of related survey works, and our distinction. Finally, Section IX is reserved for the conclusion and summary.

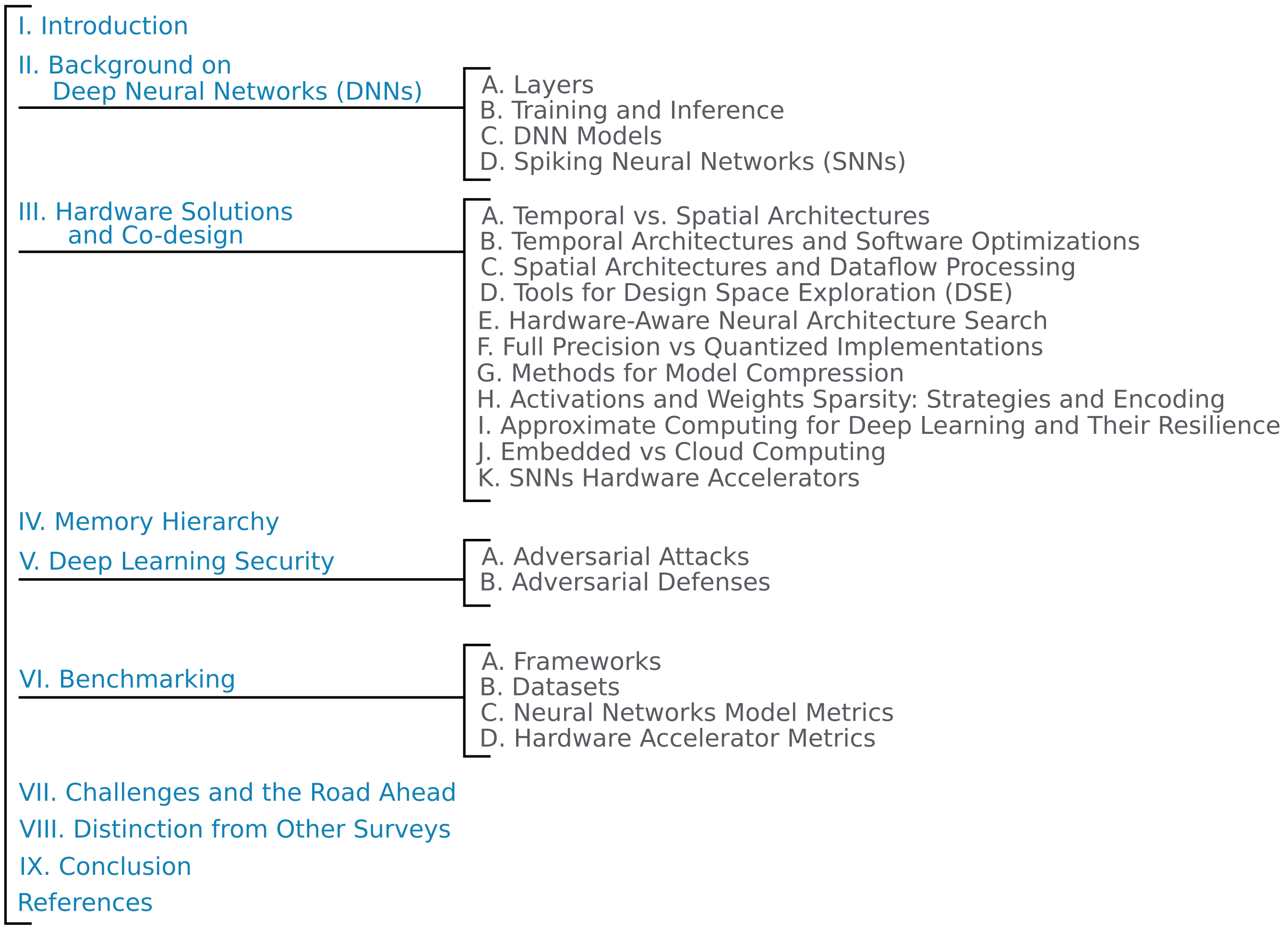
TABLE 1. List of Acronyms.

A	Activation
AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
BC	Binary Connect
BLAS	Basic Linear Algebra Subroutines
BN	Batch Normalization
BNN	Binarized Neural Network
BWN	Binary Weight Net
CIS	Compressed Image Size
CNN	Convolutional Neural Network
Conv	Convolutional
CP	Canonical Polyadic
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
DL	Deep Learning
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
FC	Fully Connected
FFT	Fast Fourier Transform
FM	Feature Map
FPGA	Field Programmable Gate Array
GAN	Generative Adversarial Network
GD	Gradient Descent
GLB	Global Buffer
GeMM	General Matrix Multiplication
GPU	Graphic Processing Unit
IFM	Input Feature Map
IoT	Internet-of-Things
L	Loss
LIF	Leaky-Integrate-and-Fire
LIM	Logic-in-Memory
MAC	Multiply-and-Accumulate
ML	Machine Learning
MSE	Mean Squared Error
NAS	Neural Architecture Search
NN	Neural Network
NoC	Network-on-Chip
NPU	Neural Processing Unit
OFM	Output Feature Map
OS	Output Stationary
PE	Processing Element
QWN	Quatized Neural Network
ReLU	Rectified Linear Unit
RF	Register File
RLC	Run Length Coding
RS	Row Stationary
SIMD	Single-Instruction Multiple-Data
SIMT	Single-Instruction Multiple-Threads
SNN	Spiking Neural Network
SRAM	Static Random Access Memory
STDP	Spike Time Dependent Plasticity
TPU	Tensor Processing Unit
TTFS	Time To First Spike
TWN	Ternary Weight Net
VLSI	Very Large Scale Integration
W	Weight
WS	Weight Stationary

II. BACKGROUND ON DEEP NEURAL NETWORKS (DNNs)

The constituent element of a neural network is the *neuron*, also called *perceptron*, a computational block that attempts to model the behavior of a biological neuron, which is shown in Figure 3.

A biological neuron consists of the cell body (soma), the dendrites and an axon [25]. The dendrites and the axon are filaments; the former receive stimuli, that are then processed by the soma, while the latter takes the neuron output signal to other neurons. Neurons are electrically excitable; when the input voltage exceeds a certain threshold, a pulse, called *action potential*, is generated on the axon. The neuron's response is *all-or-none*, i.e., the neuron can only have no response or full response depending on the input voltage value. The computational model adopted in artificial neural



I. Introduction

II. Background on Deep Neural Networks (DNNs)
- A. Layers
- B. Training and Inference
- C. DNN Models
- D. Spiking Neural Networks (SNNs)

III. Hardware Solutions and Co-design
- A. Temporal vs. Spatial Architectures
- B. Temporal Architectures and Software Optimizations
- C. Spatial Architectures and Dataflow Processing
- D. Tools for Design Space Exploration (DSE)
- E. Hardware-Aware Neural Architecture Search
- F. Full Precision vs Quantized Implementations
- G. Methods for Model Compression
- H. Activations and Weights Sparsity: Strategies and Encoding
- I. Approximate Computing for Deep Learning and Their Resilience
- J. Embedded vs Cloud Computing
- K. SNNs Hardware Accelerators

IV. Memory Hierarchy

V. Deep Learning Security
- A. Adversarial Attacks
- B. Adversarial Defenses

VI. Benchmarking
- A. Frameworks
- B. Datasets
- C. Neural Networks Model Metrics
- D. Hardware Accelerator Metrics

VII. Challenges and the Road Ahead

VIII. Distinction from Other Surveys

IX. Conclusion

References

FIGURE 2. **Paper outline.**

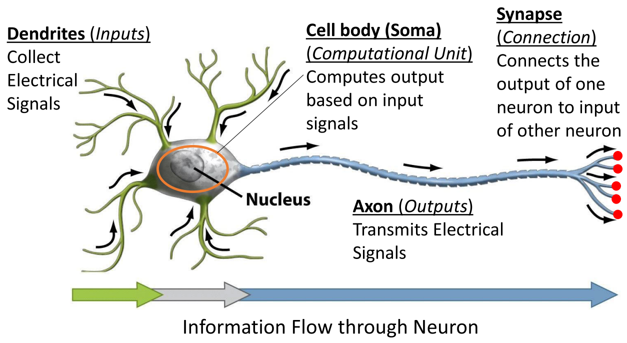

Dendrites (Inputs)
Collect Electrical Signals
Cell body (Soma)
(Computational Unit)
Computes output based on input signals
Synapse
(Connection)
Connects the output of one neuron to input of other neuron
Nucleus
Axon (Outputs)
Transmits Electrical Signals
Information Flow through Neuron

FIGURE 3. **Model of a biological neuron, adapted from [25].**

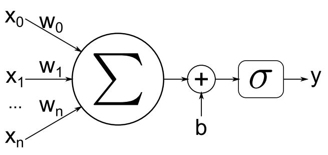


FIGURE 4. **Model of an artificial neuron.**

networks has been modified in time [26], [27] until reaching the configuration now adopted (Figure 4). In essence, it performs a weighted sum of all its inputs (Eq. 1), to which a bias term b is added to include a possible offset. The output of the neuron is then obtained applying a non-linear function $\sigma(\cdot)$ (Eq. 2).

$$g(\mathbf{x}) = \sum_{n=0}^{N-1} \mathbf{x}[n]\mathbf{w}[n] \tag{1}$$

$$y = \sigma\left(g(\mathbf{x}) + b\right) \tag{2}$$

Artificial neural networks are constructed as directed graphs whose nodes represent the neurons. If the graph is acyclic, the network is a *feedforward* NN. If the graph is cyclic, the network is *recurrent* and has a temporal dynamic behavior.

As shown in Figure 5, the nodes are organized in layers: in a feedforward NN, each neuron of layer l receives its inputs from layer $l-1$ and sends its activation to the neurons of layer $l+1$. The inputs to the network form the *input layer*, and there is at least one layer that processes the input, which is called *output layer*. All the layers inserted between the input and output layers are defined as *hidden layers*. The number of hidden layers determines the *depth* of a neural network. If there are more than three hidden layers, the neural network is typically called a *Deep Neural Network*

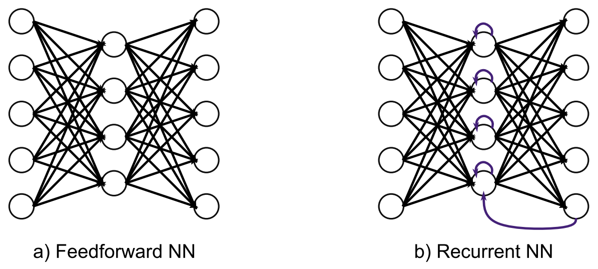


FIGURE 5. Examples of (a) a feedforward NN and (b) a recurrent NN.

```
for (n=0; n<N; n++)
  for (co=0; co<Co; co++)
    for (ci=0; ci<Ci; ci++)
      O[n][co] += I[n][ci] * W[co][ci]
```

FIGURE 6. Pseudocode of an FC layer.

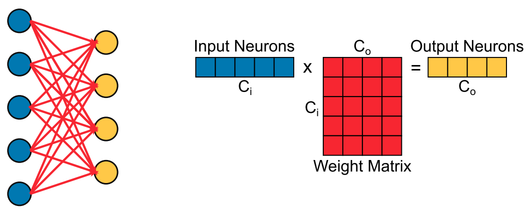


FIGURE 7. Example of an FC layer (left) and of how it can be modeled by a vector-matrix multiplication (right).

(DNN) [28]. An NN learns how to solve different problems by finding the optimal values for the weights and the biases of its neurons, that can be organized and connected in different ways, as discussed in the following section.

A. LAYERS

1) FULLY CONNECTED (FC) LAYERS

In a Fully Connected layer, each neuron of layer l receives as inputs all the activations of layer $l-1$, therefore, each output neuron performs a weighted sum of all the input neurons:

$$\mathbf{O}[c_o] = \sum_{c_i=0}^{C_i-1} \mathbf{W}[c_o, c_i]\mathbf{I}[c_i] + \mathbf{b}[c_o]$$
$$0 \le c_o < C_o, \quad 0 \le c_i < C_i$$

where C_i and C_o are the number of neurons of layers $l-1$ and l respectively. Figure 6 shows the pseudocode that implements an FC layer. In Figure 6 N is the *batch size*, where a *batch* is a collection of inputs that can be processed in parallel.

From the equation and the pseudocode that describes it, it is possible to see that an FC layer is a vector-matrix multiplication with the weights arranged in a $C_i \times C_o$ matrix (see Figure 7).

Since C_i and C_o can assume high values, the number of parameters of an FC layer is potentially huge. However, it is not always necessary for an output neuron to receive information from all the input neurons. For this reason, Convolutional layers have been introduced.

2) CONVOLUTIONAL (CONV) LAYERS

FC layers are not well suited for tasks like object detection and recognition since their high degree of connectivity leads to an explosion of the number of parameters required to deal with high-resolution images. Moreover, FC layers treat inputs that are close together or far apart equivalently, ignoring the spatial structure present in images. To overcome these two problems, in 1998 a new architecture was proposed [29], known as Convolutional Neural Network (CNN), that includes Conv layers and exploits the ideas of *local receptive fields* and *shared weights*. The idea of local receptive fields has its biological counterpart in the study of David H. Hubel and Torsten Wiesel [30] on the visual cortex of a cat. They demonstrate that some neurons are activated when the cat is visually exposed to vertical lines, while different neurons respond to lines oriented along different angles. There are thus *locally sensitive* neurons that are sensitive to a small portion of the visual field and higher-level neurons that are sensitive to larger portions and therefore analyze more complex patterns. Adapting the same idea to a neural network, the neurons are organized in a 2D grid, i.e., a *feature map*, and a neuron of layer l does not receive all the activations of the layer $l-1$, but it is instead connected to a small receptive field of dimension $[H_k \times W_k]$. The size of the receptive field and consequently of the weight matrix is commonly referred to as *kernel size* and the distance between adjacent receptive fields is defined by a stride parameter S. Applying the idea of shared weights, all the neurons of layer l have the same matrix of weights, detecting the same feature in different locations of layer $l-1$. To detect multiple features, a Conv layer has multiple channels, i.e., there are multiple feature maps.

The computations performed in a Conv layer involve an *input feature map* **Ifm** of size $[C_i \times H_i \times W_i]$, the *weights* **W** of size $[C_i \times C_o \times H_k \times W_k]$, and a *bias term* **b** of size $[C_o]$. The result of the computation is an *output feature map* **Ofm** of size $[C_o \times H_o \times W_o]$, computed as follows:

$$\begin{aligned}\mathbf{Ofm}[c_o, h_o, w_o] &= \sum_{c_i=0}^{C_i-1}\sum_{h_k=0}^{H_k-1}\sum_{w_k=0}^{W_k-1} \\ &\times \Big(\mathbf{W}[c_i, c_o, h_k, w_k]\mathbf{Ifm}[c_i, Sh_o+h_k, Sw_o+w_k]+\mathbf{b}[c_o]\Big)\end{aligned}$$
$$0 \le c_o < C_o, \quad 0 \le h_o < H_o, \ 0 \le w_o < W_o$$
$$0 \le h_k < H_k, \quad 0 \le w_k < W_k$$

Figure 8 shows the pseudocode of a Conv layer, and Figure 9 gives a graphical representation.

3) POOLING LAYERS

Pooling layers are commonly placed after a Conv layer. Pooling layers have receptive fields, similarly to Conv layers. For the group of neurons in each receptive field, they return a single value that contains a statistic of the group, e.g., the maximum or the average value, as shown in Figure 10.

```
for ($n=0$; $n<N$; $n$++)
 for ($c_o=0$; $c_o<C_o$; $c_o$++)
  for ($c_i=0$; $c_i<C_i$; $c_i$++)
   for ($h_o=0$; $h_o<H_o$; $h_o$++)
    for ($w_o=0$; $w_o<W_o$; $w_o$++)
     for ($h_k=0$; $h_k<H_k$; $h_k$++)
      for ($w_k=0$; $w_k<W_k$; $w_k$++)
       $Ofm[n][c_o][h_o][w_o]$ +=
       $Ifm[n][c_i][h_o+h_k][w_o+w_k]$ * $W[c_o][c_i][h_k][w_k]$
```

FIGURE 8. Pseudocode of a Conv layer.

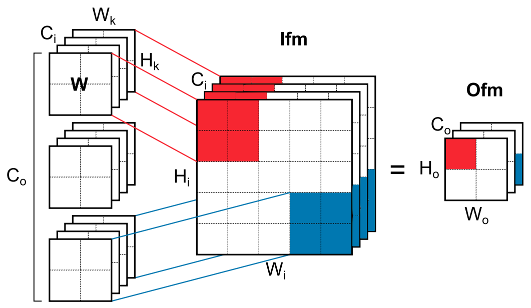


FIGURE 9. Graphical representation of the convolution operation in a Conv layer. A sub-tensor of Ifm (red) is multiplied by a sub-tensor of W and the results are accumulated to produce a single value (red) of the Ofm.

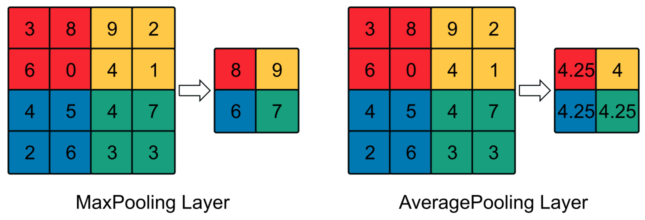


FIGURE 10. Examples of MaxPooling (left) and AveragePooling (right) layers.

The stride parameter is usually set equal to the dimension of the receptive field to have non-overlapping windows.

Pooling layers reduce the number of activations of a layer, and consequently decrease the memory requirements and the number of computations to be performed after. Moreover, pooling layers achieve invariance to small local translations. The outputs of Conv layers depend heavily on the position of the input, so even for minor variations of the inputs, there are significant variations of the outputs. Pooling layers downsample the outputs, making them more robust to small input variations.

4) NORMALIZATION LAYERS

The inputs to neural networks are usually preprocessed to have a normal distribution, i.e., zero mean and unit variance. Normalization is beneficial because it keeps different inputs in the same range of values, making them easier to analyze by the same model. Also, as will be seen in the following paragraph, layers sometimes use saturating non-linear functions, such as Sigmoid or Softmax. So having values centred on zero avoids early-saturation of activations. To apply the same normalization constraint that applies to the inputs to internal activations, Normalization layers are inserted between Conv and FC layers. It must also be noted that activations normalization speeds up the training, as the layers do not need to adapt to different distributions at each training step.

The commonly adopted normalization method is Batch Normalization (BatchNorm) [31] (Eq. 3). The operation performed by the BatchNorm layer is standardization:

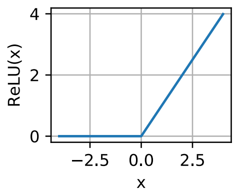
$$y = \frac{x - E[x]}{Var[x] + \epsilon} \cdot \gamma + \beta \tag{3}$$

where $E[x]$ and $Var[x]$ are the mean and standard deviation of the input tensor x, respectively. ϵ is a value necessary for numerical stability, and γ and β are two trainable parameters for the integration of the BatchNorm layer in the training process.

5) NON-LINEAR ACTIVATION FUNCTIONS

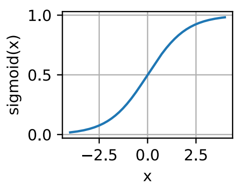
Without a non-linear activation function, the NN would be a simple cascade of linear algebra operations, unable to solve complex non-linear problems. For this reason, different non-linear functions are applied to the weighted sum of the inputs of a neuron. Some of the most popular functions are:

- *Rectified Linear Unit (ReLU)* function forces the activations to be greater than or equal to zero. It is prevalent as it is computationally efficient since it requires a simple comparison between x and 0.



$$y = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

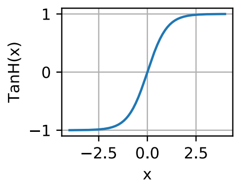
  There are some variants of the ReLU function, such as *Leaky-ReLU* or *Exponential Linear Unit (ELU)*. The former has a negative slope for values $x < 0$; the latter uses a log curve when $x < 0$. These variants have been introduced to solve the *dying ReLU* problem, i.e., since the slope of the ReLU for $x < 0$ is zero, the neurons in this region are not trained. Moreover, Leaky-ReLU and ELU are more balanced towards zero if compared to ReLU, and this helps to speed up the training.
- *Sigmoid* function normalizes the output in the range (0, 1). Contrarily to the ReLU function, it is computationally expensive, as it can be seen from its equation:



$$y = \frac{1}{1 + e^{-x}}$$

- *Hyperbolic Tangent* function (TanH) is the equivalent of Sigmoid function to bound activations in the range

(−1, 1), to model outputs that can assume negative values too.



$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- *Softmax* function is also know as *normalized exponential function*. It receives a vector of N numbers as an input: each number is normalized in the range (0, 1) and the sum of all N numbers is equal to 1. This function is used mainly in output layers if the outputs represent the classification probabilities.

$$y_i = \frac{e^{x_i}}{\sum_{j=0}^{N-1} e^{x_j}} \quad \text{for } i = 0, 1, \ldots, N-1$$

B. TRAINING AND INFERENCE

A neural network can learn to solve a problem by determining the correct values of the weights and biases of its layers: this process is referred to as *training*. Using a trained NN, with pre-learned weights and biases, is referred to as *inference*. There are different ways of training a NN (see Figure 11):

Supervised learning: It requires a set of labeled input-output pairs, i.e., a set of inputs (*data*) with the corresponding expected output (*labels*). This set of pairs is called a *training set*. During the supervised learning, the model receives a labeled input and updates its parameters based on the discrepancy between the expected output and the actual output. Supervised learning is predominantly used today in a wide range of applications, in the big-data era, thanks to the immense availability of datasets and its good performances.

Unsupervised learning: It is performed when only non-labeled data are available. It lies in finding common patterns in the data. An example of unsupervised learning is *clustering*, that clusters data based on their shared attributes. Neural networks that apply unsupervised learning are, for example, *autoencoders* and *Generative Adversarial Networks* (GANs).

Reinforcement learning: Reinforcement learning is the third main type of learning and, similar to the unsupervised learning, it does not need labeled data. The aim of reinforcement learning is the creation of autonomous agents able to make decisions in a given environment. The training scenario is composed of the agent who takes actions in an environment. There is then an interpreter who evaluates the agent's actions in terms of a reward, which is then fed back to the agent. The goal of the agent is to maximize the reward.

A supervised-learning algorithm commonly used for the training of DNNs is *gradient backpropagation*, where the input samples are fed into the network, and the outputs are computed using weights **W**. The network's outputs and expected outputs are compared, and a *loss (L)* is calculated with a *loss function*, such as Euclidean distance or Mean Squared Error (MSE). To perform the learning process, the weights are updated by a quantity proportional to the partial derivative of the loss with respect to the weights themselves, i.e., the gradient. The gradients are efficiently computed with the backpropagation algorithm, which is the *chain rule* of calculus applied to calculate the derivative of the loss starting from the output of the network and going up to the input layer.

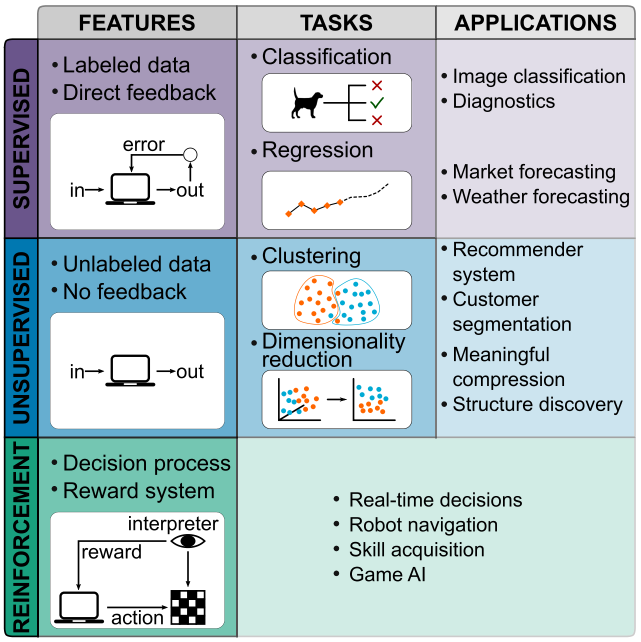


FIGURE 11. Features, achievable tasks and applications of the three existing ways of training (supervised, unsupervised, and reinforcement).

The learning actually takes place by updating the weights and biases of the network, which can be done with different *optimization algorithms*. The simplest optimization algorithm is gradient descent (GD), shown in Eq. 4, where θ is a parameter of the network and η is a scaling factor referred to as *learning rate*. Other algorithms are, e.g., GD with momentum [32], Nesterov accelerated gradient [33], Adagrad [34], Adadelta [35] and Adam [36].

$$\theta^{t+1} = \theta^t - \eta \frac{\partial L}{\partial \theta^t} \tag{4}$$

During the training of neural networks it is common to encounter the problem of *overfitting*, i.e., if a model is complex and has many parameters, it is possible that it fits the data of the training set too accurately. The model therefore ''memorizes'' the correct result for each input rather than learning to generalize, and has a poor performance on the inputs that are never seen before. The solutions to the overfitting problem are either the transition to a simpler model or employing different *regularization techniques*. L1 and L2 [37] are common regularization techniques, both require adding a regularization term to the loss function, which has the effect of reducing

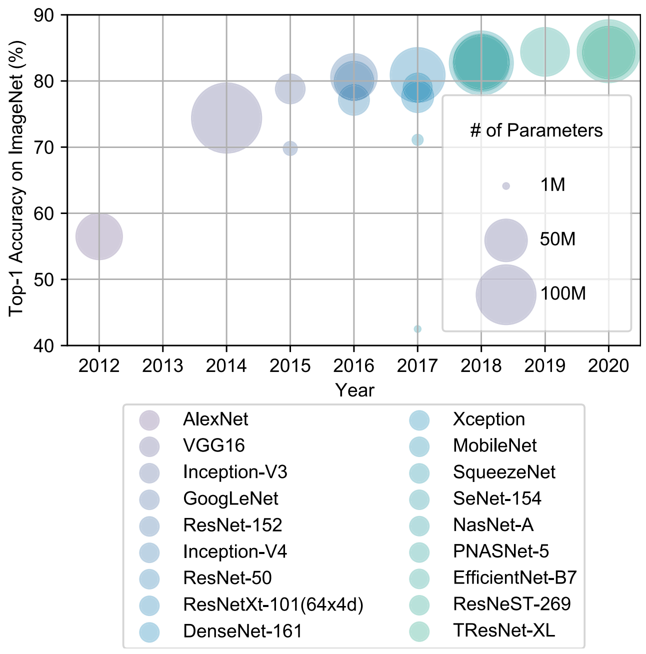


FIGURE 12. Timeline of significant neural networks models with the accuracies achieved on the ImageNet dataset [39] and the number of parameters.

the value of the weights. This results in a compressed and simpler model. Another technique that gives good results is *dropout* [38], i.e., at each iteration some neurons are randomly selected and removed from the model.

C. DNN MODELS

Over the years, many CNNs models have been proposed to achieve better to the best-possible performance for a given task. Figure 12 shows a timeline of significant neural network models with their classification accuracy in the image classification task on the ImageNet dataset [39] and number of parameters. These models will be discussed in the following paragraphs and compared in Table 2.

1) LeNet [29] (1998)

It has been one of the first neural network trained by backpropagation with a convolutional structure and has been the inspiration for the following research on CNNs. It was designed for the recognition of handwritten digits represented on 32 × 32 pixels images. LeNet-5 is a version consisting of five layers, of which the first two are convolutional layers, and the last three are FC layers. The Conv layers have 5×5 kernels and are both followed by 2×2 average pooling layers. All the layers use hyperbolic tangent (tanh) as the activation function, except the output layer that applies the softmax function.

2) AlexNet [40] (2012)

It is a CNN built for the ImageNet dataset [39], a database of more than 1.3M of high-resolution 256 × 256 pixels images divided into 1000 classes. It is the first (deep) CNN architecture to win the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC-2012) [41], achieving consistent accuracy improvements compared to the traditional non-convolutional networks winners of the previous editions. AlexNet follows the architecture of LeNet, stacking more Conv layers; it consists of five Conv layers and three FC layers. AlexNet was the first NN to introduce Rectified Linear Units (ReLU) as the activation functions, reducing the training time significantly. Moreover, to overcome the limitations imposed by the memory size of the GPUs, AlexNet adopts a parallel solution, splitting the architecture on two GPUs. To reduce the communication bottleneck, the GPUs exchange data in two points of the network only.

TABLE 2. Comparison of the models presented in Section II-C.

Model	Year	Contribution
LeNet [29]	1998	-First popular CNN
AlexNet [40]	2012	-First CNN winner of ILSVRC -Introduction of ReLU
VGG16 [42]	2014	-Smaller kernels
GoogLeNet [43]	2015	-Inception block
ResNet [44]	2016	-Skip connections -Residual learning
ResNetXt-101_64x4d [45]	2017	-Grouped convolution
DenseNet161 [46]	2017	-Regular structure
CapsNet [47]	2017	- Capsules - Dynamic Routing
SeNet154 [48]	2018	-Dependencies between feature maps are exploted
NasNet-A [49]	2018	- Neural Architecture Search - Transfer learning

3) VGG [42] (2014)

Thanks to the availability of hardware resources supporting heavier computations, the initial trend in NN research has been the design of deeper and deeper architectures. VGG is a model that takes the structure of AlexNet and furtherly increases the number of Conv layers. In particular, VGG-16 has 13 Conv layers and three FC layers, while VGG-19, with a total of 19 layers, was the winner of ILSVRC-2014.

4) GoogLeNet [43] (2015)

It is based on the intuition of finding a dense structure, i.e., an inception module, and then building the network by stacking these modules. An inception module (see Figure 13) captures features at various scales and concatenates them at the output, passing to the next layer different levels of information. The increase of the depth of the NNs has allowed to improve their accuracy but has however led to the appearance of the *vanishing gradient problem*. Since during backpropagation the gradients are computed with the chain rule and the values are often in the range [0, 1] or [−1, 1], the magnitude of the gradients decreases exponentially with the depth of the network. In the earlier layers, the gradients can become so small that they prevent the correct training. To overcome

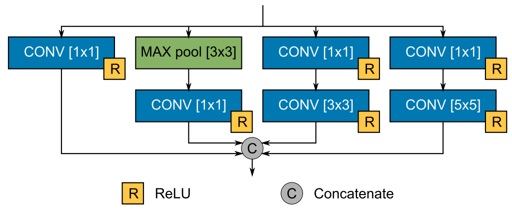


FIGURE 13. Inception Module used in GoogLeNet [43]. Convolutions are performed in four parallel branches and the four outputs are concatenated to produce the single output of the Inception module.

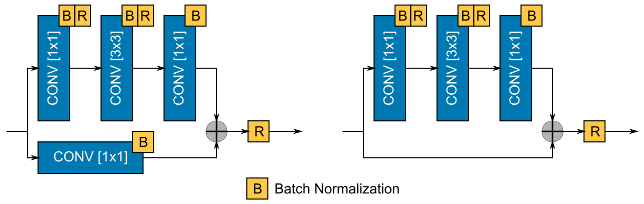


FIGURE 14. Skip connection modules used in Residual Networks [44]. Three convolution are performed in series and a parallel connection is added. In the parallel connection, it is possible to choose between a 1 × 1 convolution (left) or the identity function, i.e., no operation (right). The results of the two branches are summed.

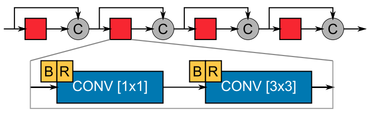


FIGURE 15. Dense blocks used in DenseNets [46]. The output and input of each blue block, that is the series of two convolutions, are concatenated.

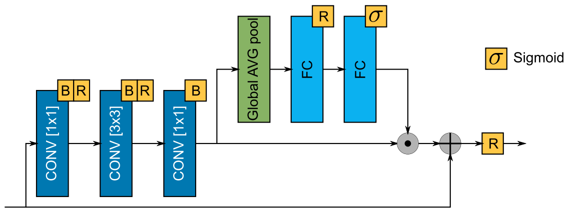


FIGURE 16. Residual module as modified in Squeeze-and-Excitation Networks [48]. A skip connection is inserted in parallel to a pooling and two FC layers, and the output of the two branches are multiplied. As in traditional residual modules, a skip connection runs in parallel to the whole block.

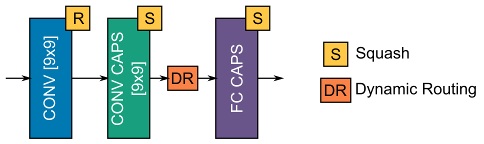


FIGURE 17. Capsule Network [47].

this problem, GoogLeNet has two additional classifiers used for training only that take the activations at earlier stages of the network, and therefore increase the magnitude of their gradients. GoogLeNet successors are Inception-v3 [50] and Inception-v4 [51].

5) ResNet [44] (2015)

To work around the vanishing gradients problem, Residual Networks (ResNets) have adopted and made popular *skip connections*, shown in Figure 14, that run in parallel to a series of Conv layers and avoid excessive degradation of the gradients during backpropagation. Moreover, ResNets are the first architectures to use batch normalization layers. Based on ResNet architecture, different models with higher accuracies have been proposed over the years, such as ResNetXt [45], ResNeST [52], or TResNet [53].

6) DenseNet [46] (2016)

Given the success of skip connections, DenseNets adopt a regular and therefore simpler connection pattern. As shown in Figure 15, in a Dense Block, every layer receives in input a concatenation of the activations of all the preceding layers. A DenseNet is then built by stacking Dense Blocks of different depth, interleaved by Conv and Pooling layers for dimensionality reduction.

7) SENet [48] (2017)

Squeeze-and-Excitation Networks (SENets) modify traditional layers, e.g., Conv layers, or blocks, e.g., inception or residual modules, to model the relationship between the different channels of the feature maps. Figure 16 shows how a residual module is modified following the SE approach. SENet-154 is the NN winner of ILSVRC-2017, which is built integrating SE blocks in a version of ResNetXt [45].

8) CAPSULE NETWORK (2017)

The Capsule Networks were created in a try to solve some of the problems of CNNs, such as the loss of data caused by pooling layers or the high sensitivity to input shifts or rotations. The idea of *capsules* was introduced in [54] and the first network model was proposed in [47]. In [47], the neurons are replaced by capsules, i.e., a vector of neurons. Each element of the vector encodes an instantiation parameter of an entity, e.g., the width or the rotation, and the length of the vector represents the instantiation probability of the entity. Since the length of the vector represents a probability, its value must be in the range [0, 1]. For this reason, the *squash* function (Eq. 5) is used as non-linear activation function in the capsule layers.

$$\vec{y} = \frac{|\vec{x}|^2}{1 + |\vec{x}|^2} \frac{\vec{x}}{|\vec{x}|} \tag{5}$$

Moreover, in Capsule Networks, the pooling layers are substituted by a *dynamic routing algorithm* that strengthens the connections between capsules of adjacent layers if relevant entities are detected. Figure 17 shows the Capsule Network model as proposed in [47]. The work in [55] proposes instead a model in which the values of the capsules are arranged in matrices, and the dynamic routing is substituted by the *EM routing*.

9) NASNet [49] (2018)

NASNet is the first popular neural network model obtained with neural architecture search. The approach of NasNet is the search of a cell for a simple dataset in a small search

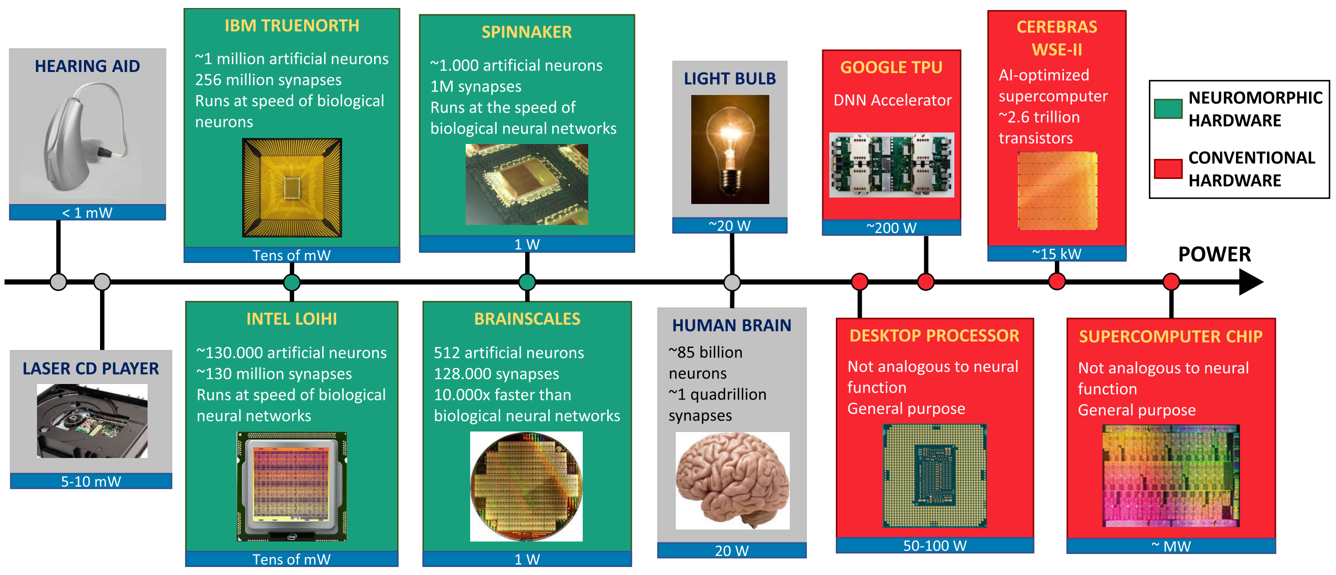


FIGURE 18. Comparison of power consumption between conventional hardware architectures and neuromorphic architectures.

space. The cells can then be stacked to work on more complex datasets. Other models resulting from neural architecture search are PNASNet-5 [56] and EfficientNet [57].

D. SPIKING NEURAL NETWORKS (SNNs)

Recently, Spiking Neural Networks (SNNs), considered as the third generation of neural networks [58], have received an increasing interest in the fields of deep learning and neuroscience, because of their extremely energy-efficient nature. SNNs, in contrast to the traditional DNNs, base their computational models much closer to that of the biological neurons, with a spike-based communication mechanism [59]. Due to their bio-inspired computations, SNNs bear a high potential to be the most promising solution for bridging the energy efficiency gap between the artificial machines and the human brain. A custom SNN hardware support is provided by neuromorphic computing, a relatively novel branch of computer architecture. The underlying goal is to reproduce in hardware the same computations that are executed in the human brain. Some examples of state-of-the-art neuromorphic designs, like IBM TrueNorth [60], SpiNNaker [61], BrainScale [62] and Intel Loihi [63], will be discussed in Section III-K. Figure 18 compares several hardware architectures, showing how efficient in terms of power consumption are neuromorphic solutions, compared to conventional designs [64]. Moreover, another energy efficiency benefit in the neuromorphic research comes from the new sensor data formats. For instance, the event-based sensors such as the dynamic vision sensor (DVS) cameras [65] resemble the behavior of the human retina, in such a way that spikes are generated only when movements of the recorded subjects are detected.

1) SPIKING NEURON MODELS

Modeling a spiking neuron is a challenging task. These models must be at the same time biologically accurate and computationally simple. When an input spike arrives to the

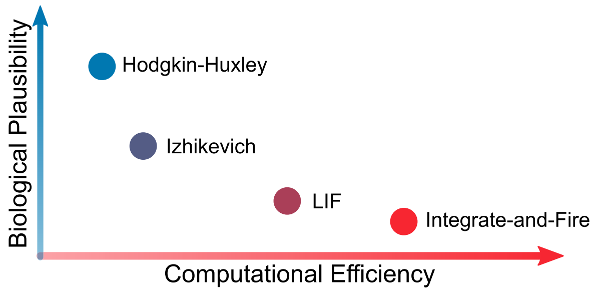


FIGURE 19. Comparison of different spiking neuron models.

neuron, the associated synaptic weight w_i is integrated on the membrane and, consequently, the neuron membrane potential V_m is increased. When the membrane potential overcomes a threshold V_t, the neuron fires, emitting a spike at the output, and its membrane potential is reset to a value V_R. Moreover, the membrane potential decreases continuously through time due to a leakage, according to a leak rate τ_m between spikes.

Different spiking neuron models have been proposed in the literature. Figure 19 shows the trade-off between biological plausibility and complexity of these models. The Hodgkin-Huxley model [66] is very biologically-plausible, but extremely computational intensive. The Izhikevich model [67] is slightly less complex, but still very computational intensive. On the other end, the Integrate-and-Fire is too simple and not very accurate in terms of biological plausibility. The most commonly adopted model is the Leaky-Integrate-and-Fire (LIF) [68], which is relatively simple and also takes into account the membrane leakage.

2) SPIKE ENCODING

In order to provide input spikes and to collect the resulting output spikes of the SNN, the information has to be properly

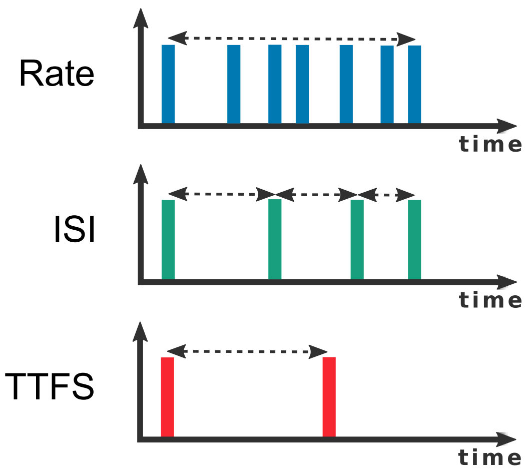


FIGURE 20. Comparison between Rate, Inter-spike interval (ISI), and Time to first spike (TTFS) encoding techniques for SNNs.

coded using spikes. Different approaches used to obtain such a conversion [69] are shown in Figure 20:

- **Rate coding**: the information is coded as the mean firing rate of the generated spikes in a defined observation period.
- **Inter-spike interval (ISI)**: the intensity of the activation is coded as the precise delay between consecutive spikes.
- **Time to first spike (TTFS)**: the information is encoded in the latency that goes from the beginning of the stimulus to the time of the first output spike. This solution enables a very fast information processing, carrying enough information.

3) SNN TRAINING

Regarding the SNN training algorithm, the different possibilities have been explored are summarized in Figure 21. For unsupervised learning, the possible algorithms are Hebbian Learning [70], the **Spike-Time-Dependent Plasticity (STDP)** [71], [72], and the Spike-Driven Synaptic Plasticity (SDSP) [73]. The most widely adopted method is the STDP, which is based on temporal relations between the *presynaptic* spikes (at the input of the neuron) and the *postsynaptic* spikes (at the output of the neuron). Basically, the synaptic weight is tuned accordingly to the temporal correlation between the presynaptic and postsynaptic spikes. The STDP algorithm can be optimized through the FSpiNN framework [74], for executing energy-efficient SNNs on edge devices.

For supervised learning, a fundamental challenge arises, because the traditional learning method, i.e., the backpropagation, cannot be applied due to the non-differentiability of the SNN loss function [75]. Hence, two different procedures can be adopted to solve or bypass the problem, thereby achieving supervised learning for SNNs:

1) *Approximate the derivative of the spike trains.* This solution has been extensively studied in the works of [76], [77] [78], [79] [80], [81] [82], [83] [84], [85], which provide different types of approximations. The advantage is that the network can learn based on the temporal information of the spikes. For example, DECOLLE [86] introduces a local learning rule for continuous SNN learning. On the other hand, with this approach it is challenging to match the consolidated state-of-the-art high accuracy results of the DNNs.
2) *Train a DNN offline and convert it to SNN.* This approach [87] allows to use the most advanced training policies and techniques for DNNs. An efficient conversion [88] requires a comprehensive study of different conversion parameters to adapt the DNN-to-SNN conversion process to the neuromorphic hardware platform. The main drawback is that a certain accuracy drop is encountered during the conversion. To overcome this, the recent work of [89] proposed a hybrid approach consisting of converting the DNN to SNN ad then incrementally training the SNN with an approximated backpropagation. Moreover, the max pooling operations cannot be implemented with spike rates [90]. Therefore, max pooling layers are replaced by average pooling, which is easy to implement but shows an accuracy drop.

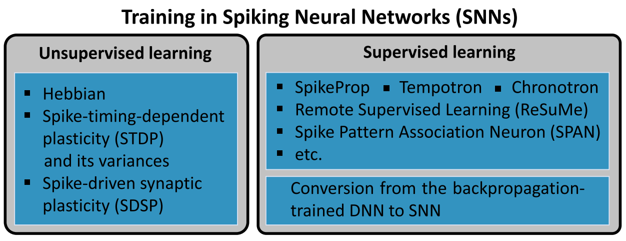

Training in Spiking Neural Networks (SNNs)
Unsupervised learning
Hebbian
Spike-timing-dependent plasticity (STDP) and its variances
Spike-driven synaptic plasticity (SDSP)
Supervised learning
SpikeProp
Tempotron
Chronotron
Remote Supervised Learning (ReSuMe)
Spike Pattern Association Neuron (SPAN)
etc.
Conversion from the backpropagation-trained DNN to SNN

FIGURE 21. Training techniques for SNNs.

III. HARDWARE SOLUTIONS AND CO-DESIGN

A. TEMPORAL VS SPATIAL ARCHITECTURES

Neural networks are a class of algorithms with an inherent parallelism. Two types of parallelism can be identified [91]. The neuron and consequently the FC and Conv layers have a *topological parallelism* since the Multiply-and-Accumulate (MAC) operations that they perform have no data dependencies and can be executed in parallel. Moreover, the training sets consist of a large number of samples, that rather than being processed one at a time can be fed into the network in batches (*operational parallelism*).

The intrinsic parallelism of the layers can be exploited using parallel computing paradigms to increase the performance of the hardware implementations of NNs. Among the various solutions for parallel computation, temporal and spatial architectures [92] are distinguished. Both the architectures consist of a large number of Processing Elements (PEs) that perform operations in parallel on the same or different data. In temporal architectures, the PEs can only access data from the central memory, the control is centralized, and there are no inter-PEs connections. In spatial architectures, on the contrary, each PE can also have its control

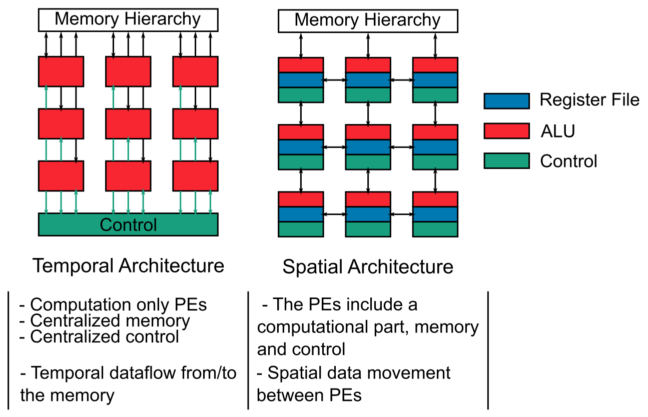


FIGURE 22. **Basic models of temporal (left) and spatial (right) architectures.**

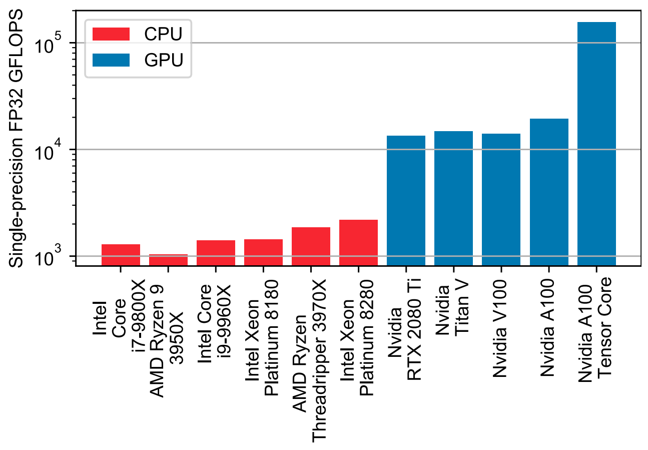


FIGURE 23. **GFLOPS comparison between different CPUs and GPUs.**

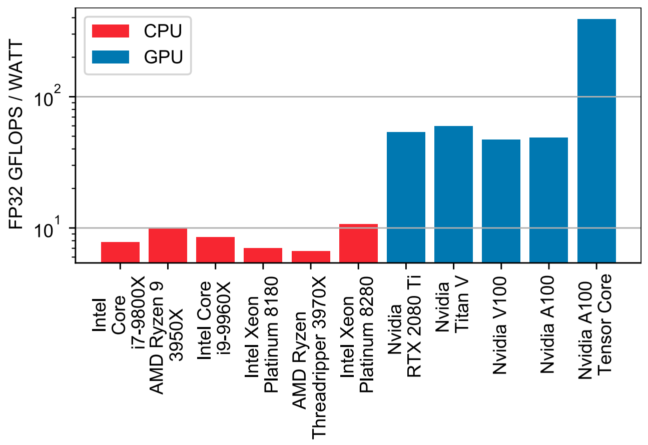


FIGURE 24. **GFLOPS/WATT comparison between different CPUs and GPUs.**

logic and one or more local memory locations. Most importantly, in spatial architectures, the PEs are interconnected to exchange data with each other, creating a processing array. Figure 22 shows the differences between temporal and spatial architectures.

In the following, subsections III-B and III-C describe temporal and spatial architectures in detail respectively, and how to efficiently deploy neural networks on them.

B. TEMPORAL ARCHITECTURES AND SOFTWARE OPTIMIZATIONS

Temporal architectures are commonly adopted in general-purpose platforms, such as CPUs and GPUs. CPUs can nowadays be realized as *vector processors* (e.g., Intel's Xeon Phi x200 and Skylake-X CPUs) with an ability of working with multiple data elements simultaneously rather than with a single data at a time. Vector processors have multiple Arithmetic Logic Units (ALUs) that work synchronously and perform an instruction on a vector of data. Therefore, vector processors use the Single-Instruction-Multiple-Data (SIMD) technique. Among the available hardware platforms, CPUs are often the least used for DNNs inference or training, as they provide lower FLOPS and FLOPS/WATT performance (see Figures 23 and 24). However, manufacturers have recently undertaken measures to accelerate the deployment of NNs on CPUs. For example, at the instruction level, Intel has added the AVX-512 Vector Neural Network Instructions (AVX-512 VNNI) to the AVX-512 Instruction Set [93] to accelerate CNNs. In addition, Intel announced that the next generation of Cooper Lake and Sky Lake processors will support Brain Floating Point (*bfloat16*) operations [94]. *bfloat16* is a floating-radix-point format on 16 bits with a dynamic range comparable with the dynamic range of the 32-bit IEEE 754 floating-point format. *bfloat16* is also supported by ARMv8.6-A and AMD's ROCm library. Intel has also created BigDL [95], an ML library for the distributed acceleration of DNN algorithms on CPU clusters.

GPUs are manycore architectures with up to thousands of cores that are specifically designed for parallel computation (e.g., 5120 cores in Nvidia V100 GPU [96]). Similarly to vector CPUs, GPUs adopt the Single-Instruction-Multiple-Thread (SIMT) execution model, first introduced by Nvidia. The SIMT model executes a single instruction simultaneously on multiple cores. Each core receives a different data that belongs to multiple threads running in parallel. GPUs are the real workhorses for DNNs training in particular, and in certain cases for inference as well. Among the various GPU manufacturers, Nvidia has put a lot of emphasis on GPU hardware and software optimization for DL. Most DL frameworks support the execution on Nvidia GPUs, e.g., Pytorch [97], Tensorflow [98], or Caffe [99]. One of the great advantages of Nvidia GPUs is cuDNN [100], a highly optimized library of primitives for DNNs. cuDNN is not the only library for DL, rather all Nvidia libraries for DNN/ML are collected in CUDA-X AI [101]. In the latest high-end GPUs, Nvidia has combined traditional CUDA Cores with Tensor Cores [96], which are optimized for large matrix operations. Tensor Cores can also support mixed-precision operations. In the new Nvidia A100, the Tensor Cores support a new format, the Tensor Format (TF32), with which performance is 10x higher when compared to the performance of the

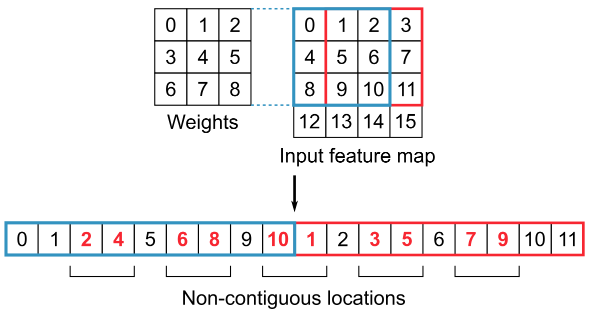


FIGURE 25. Discontinuity in memory accesses when performing convolution on a matrix stored by rows.

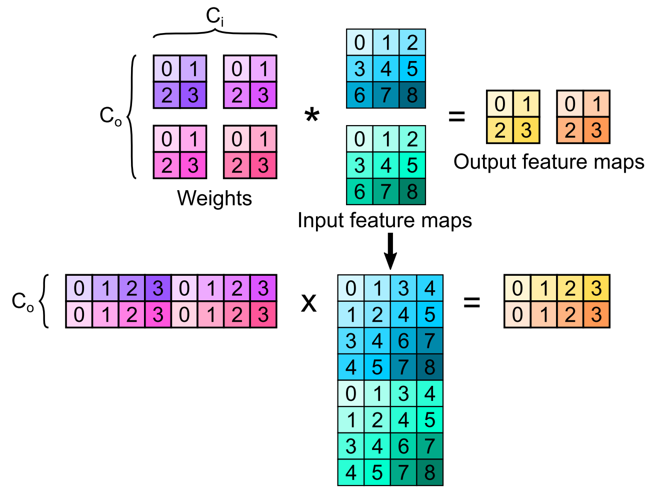


FIGURE 26. Convolution lowering: mapping a convolution to a matrix-matrix multiplication by rearranging the matrices.

FP32 format on the V100 architecture [102]. In addition, Nvidia A100's Tensor Cores can also take advantage of the sparsity of tensors, very common in DNNs, to achieve up to 2x higher performance.

At the software level, several libraries have been created to optimize *Basic Linear Algebra Subroutines* (BLAS) on both CPUs (e.g., AMD Core Math Library (ACML), Intel Math Kernel Library (Intel MKL) or OpenBLAS) and GPUs (e.g., Nvidia cuBLAS or Intel clBLAS). Among the numerous subroutines implemented, the BLAS also include element-wise matrix multiplication, matrix-vector multiplication and matrix-matrix multiplication, also called General Matrix Multiplication (GeMM). For what concerns neural networks the BLAS come in hand for the FC layer that, as explained in Section II-A, can be seen as a vector-matrix multiplication or as a matrix-matrix multiplication in case of batched computation.

Optimizing the computation of the Conv layers is a more challenging task. The operations between a weight kernel and the subsets of the input feature maps are simple point-wise multiplications of matrices, but the memory access pattern is complex. Figure 25 shows how, if an input feature map is stored by rows, it is necessary to perform accesses to non-contiguous locations of memory.

Several computational transforms have been proposed to apply the optimized BLAS to Conv layers. Many of the software libraries mentioned above lower the convolution into a GeMM as proposed in [103], [104] and shown in Figure 26. A 4D-tensor of weights is flattened to a 2D matrix, while the data in the input feature maps are duplicated and rearranged following a pattern that leads to the correct result of a convolution by performing a matrix multiplication. This method is very efficient since the GeMM routine is highly optimized. However, it requires data to be duplicated up to $H_k \times W_k$ times, with the dimension of the input feature maps moving from $C_i \times H_i \times W_i$ to $C_i H_k W_k \times H_o W_o$. This approach, therefore, requires a large memory for temporary allocation.

The GeMM method for convolution can further be optimized by applying the Strassen algorithm [105], [106] that reduces the number of necessary multiplications by partitioning the matrices. The number of multiplications is reduced of 1/8 at each partition, at the cost of a higher number of additions.

A different approach consists of transforming both the input feature maps and the weights from the space domain to the frequency domain with the FFT algorithm [107]. In the frequency domain, the convolution operation becomes an element-wise multiplication of matrices. However, the FFT algorithm introduces a high computational overhead for the domain change, and its efficiency has only been proven valid for large weight kernels and unitary strides. Another approach often used is based on the Winograd algorithm [108], [109], which, unlike the FFT algorithm, is particularly efficient for small kernels.

Direct convolution can also be performed exploiting the parallel hardware solutions offered by modern CPUs and GPUs. In [110] and [111] it is shown how to rearrange the tensors to have more efficient memory accesses, and how to perform operations to take full advantage of Intel AVX-512 [93] vector instructions.

C. SPATIAL ARCHITECTURES AND DATAFLOW PROCESSING

Spatial architectures are commonly implemented on FPGAs and ASICs, that allow for a design tailored on specific applications at the price of less flexibility. Neural networks are particularly suitable for this kind of hardware implementation since the type and order of operations of each layer is fixed and known a priori. Therefore, it is possible to develop specialized and highly optimized circuits.

The operations carried out in the neural networks are simple, mostly multiply-and-accumulate (MACs), but they must be performed on a large set of data. Therefore, the bottleneck is not caused by computation but by the memory accesses that are necessary to fetch and store the inputs and the results, respectively. Every MAC requires three data elements to be read from memory (input pixel, weight and partial

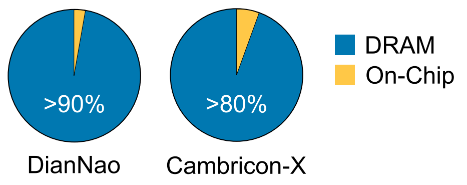


FIGURE 27. Energy breakdown of two state-of-the-art DNNs accelerators.

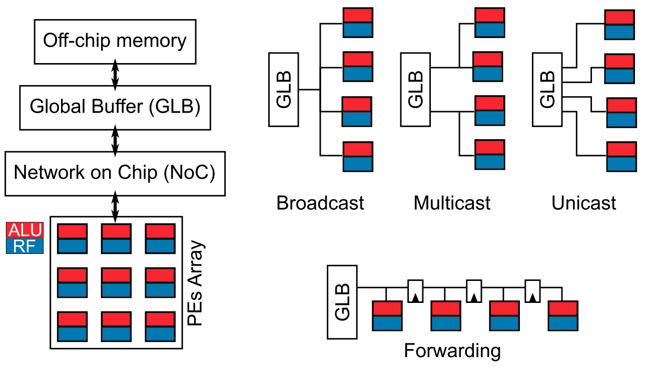


FIGURE 28. (left) General structure of a hardware accelerator for DNNs and (right) interconnection scheme.

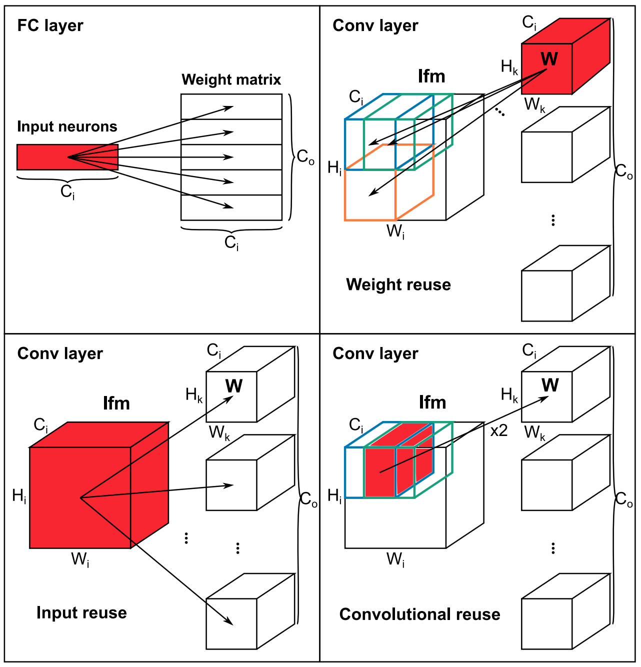


FIGURE 29. Data reuse in an FC layer and in a Conv layer.

sum) and one data element to be written (updated partial sum). It has been demonstrated that a DRAM access has an energy cost of $\sim$ 2 orders of magnitude higher than a MAC operation [112]. The enormous DRAM access energy cost compared to the computational energy has been observed in many state-of-the-art DNNs accelerators such as DianNao [24] or Cambricon-X [113] (Figure 27).

A typical hardware architecture of a DNNs accelerator (Figure 28 left) consists of:

- An **off-chip memory** (usually DRAM), to store the weights and the activations of the whole network. This level of memory can typically contain several GBs of data.
- An **on-chip global buffer (GLB)**, large enough to hold the weights and inputs necessary to feed all the PEs. The energy needed to access the GLB can be two orders of magnitude lower than that of the DRAM [114].
- An array of hundreds of **PEs**, each containing an ALU to perform MACs operations in parallel. The PEs usually also include one or more Register Files (RFs) to locally store data with an energy cost-per-access lower than that of the GLB.
- The PEs are connected with each other and to the GLB by a **Network-on-Chip (NoC)**. The data must be moved coordinately through the PEs to obtain the correct result, depending on how the operations are temporally scheduled and spatially distributed on the PEs. The NoC can then assume different configurations to implement various communication patterns, represented in Figure 28 right, depending on how data must be delivered.

Given the energy cost required by a DRAM access, the design of state-of-the-art DNNs accelerators focuses on the exploitation of *data reuse*, i.e., optimizing the architecture, the mapping of data on the PEs and the temporal scheduling of operations to maximize the reuse of data when they are stored in the lower-level memories such as the RFs or the GLB.

The different layers in an NN allow for taking advantage of various opportunities of data reuse, as explained in the following.

1) FC LAYER

A FC layer can be described as a matrix-vector multiplication and it therefore presents an opportunity for *input reuse*, since the vector of the input neurons is dot-multiplied with each row of the matrix of weights (see Figure 29).

2) CONV LAYER

The Conv layer has three different opportunities for data reuse (see Figure 29). To perform the convolution operation, a weight kernel is slid over the whole input feature map. There is an opportunity for *weight reuse* since the same weight kernel is multiplied for multiple subsets of the input feature maps. In particular, each of the C_o kernels is reused $H_o \times W_o$ times.

There is an *input reuse* opportunity too, since the input feature maps are used C_o times to generate all the output feature maps. The last reuse opportunity is defined as *convolutional reuse* [114], and it exploits the sliding window mechanism, i.e., when computing two adjacent output pixels, there is usually an intersection between the two subsets of pixels of the input feature map used, as shown in Figure 29. The width and height of the intersection depends on the dimensions of the kernels ($H_k \times W_k$) and the horizontal and vertical strides

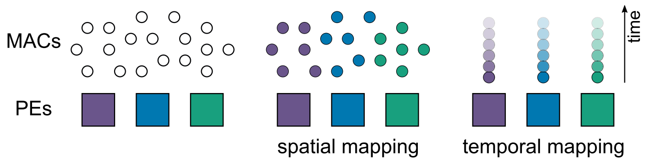


FIGURE 30. Spatial and temporal mapping of the Multiply-and-Accumulate (MACs) operations to the Processing Elements (PEs).

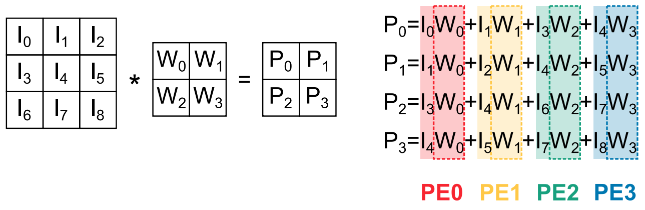

$P_0=I_0W_0+I_1W_1+I_3W_2+I_4W_3$
$P_1=I_1W_0+I_2W_1+I_4W_2+I_5W_3$
$P_2=I_3W_0+I_4W_1+I_6W_2+I_7W_3$
$P_3=I_4W_0+I_5W_1+I_7W_2+I_8W_3$
PE0 PE1 PE2 PE3

FIGURE 31. Spatial mapping of the operations in a weight stationary dataflow.

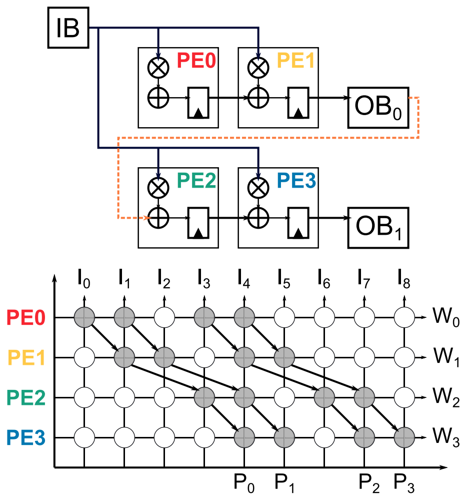


FIGURE 32. (top) PEs array and (bottom) temporal-spatial mapping of the operations in a weight stationary accelerator with inputs broadcasting and outputs forwarding.

(s_x, s_y). The convolutional reuse combines both the weight reuse and the input reuse.

3) POOLING LAYER

Pooling layers do not demand the use of weights. Therefore there are no opportunities of weight reuse. The stride parameter is usually set to have non-overlapping receptive fields, so it is not possible to exploit the sliding window mechanism for input reuse. These layers do not allow for any data reuse.

Given an array of PEs and all the MACs between weights and input feature maps that must be performed to calculate the output feature maps, each PE will execute a subset of MACs, and a number of MACs equal to the number of PEs will be executed in parallel. The MACs must, therefore, be *spatially* and *temporally mapped* to the PEs array (Figure 30). The mapping consequently defines how data must be loaded and stored from/to the memory hierarchy of the accelerator and how the NoC must be designed to correctly and efficiently deliver and collect the inputs, the weights and the partial sums. The spatial and temporal mapping of the operations is defined as *dataflow* [114].

Considering the high dimensions of the PEs array and the vast number of MACs to be computed, the space of possible mappings on a generic HW accelerator is enormous. Given the considerations on the energy consumption of the memory hierarchy, dataflows usually try to maximally exploit the opportunities of data reuse provided by the different layers of the NNs to minimize the accesses to the off-chip memory and the global buffer, and to use the data stored in the RFs as much as possible. Chen *et al.* [114] introduced a taxonomy to classify existing accelerators based on their dataflow and on how they exploit data reuse, that will be explained briefly in the following.

4) WEIGHT STATIONARY

The weight stationary dataflow aims at exploiting mainly the weight reuse to minimize the energy cost necessary to fetch the weights from the DRAM and the GLB. A subset of weights is read from the DRAM/GLB and stored in the RFs of the PEs. All the operations that involve a certain weight are then mapped to the PE where it is stored. Figure 31 shows how operations are mapped to an array of 4 PEs to perform a 2×2 convolution on a 3×3 input feature map.

Since the weights are kept stationary in the PEs, the inputs and the partial sums need to be coordinately moved through the array to optimize the data movement on the NoC too. A possibility consists of broadcasting a single input pixel of the input feature map to all the PEs and in storing each partial sum in a register then to pass it to the adjacent PE on the right. As shown in Figure 32, there are time steps in which some of the PEs perform operations that are not useful for the result (denoted in white). Moreover, the partial sums at the end of each row of processing elements needs to be stalled for $W_i - W_k$ time steps before being passed to the next row of PEs. Therefore, all of these partial sums must be stored in the GLB. The nn-X accelerator [115] allocates instead H_k FIFOs at the end of each row, each of dimension $W_i - W_k$, to introduce the proper delay.

The input pixels can be moved with the forwarding scheme to take advantage of the convolutional reuse in addition to the weight reuse. The forwarding scheme consists of placing additional registers in the PEs to store the input pixel that they receive, and to then pass it to the neighboring PEs on the right (horizontally-sliding window). Figure 33 shows a dataflow with stationary weights and input forwarding. Both in [116] and [117], H_k rows of the input feature map are processed in parallel, and the partial sums of each row are then accumulated. The inputs are therefore stored in H_k buffers, and the pixels of the input feature map are moved from the $(K - 1)$ buffer to the 0 buffer (vertically-sliding window).

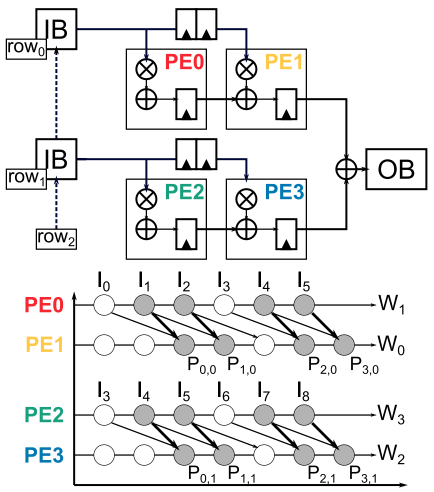
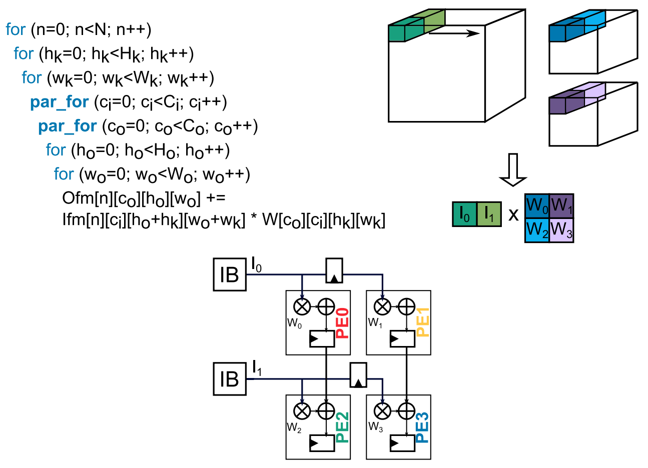
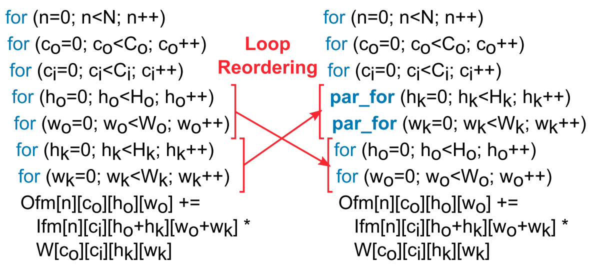


FIGURE 33. (top) PEs array and (bottom) temporal-spatial mapping of the operations in a weight stationary accelerator with inputs and outputs forwarding.



```
for (n=0; n<N; n++)
 for (hk=0; hk<Hk; hk++)
  for (wk=0; wk<Wk; wk++)
   par_for (ci=0; ci<Ci; ci++)
    par_for (co=0; co<Co; co++)
     for (ho=0; ho<Ho; ho++)
      for (wo=0; wo<Wo; wo++)
       Ofm[n][co][ho][wo] +=
       Ifm[n][ci][ho+hk][wo+wk] * W[co][ci][hk][wk]
```

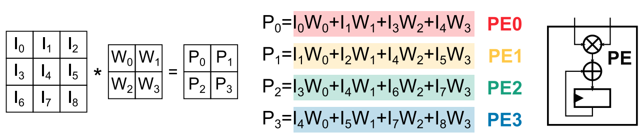
FIGURE 35. (top left) Reordered loops for a dataflow with weights stationary along dimensions C_i and C_o. (top right) Mapping the operations of a Conv layer to a matrix multiplication. (bottom) PEs array in a $C_i|C_o$ weight stationary accelerator with input and output forwarding.



```
for (n=0; n<N; n++)
 for (co=0; co<Co; co++)
  for (ci=0; ci<Ci; ci++)
   for (ho=0; ho<Ho; ho++)
    for (wo=0; wo<Wo; wo++)
     for (hk=0; hk<Hk; hk++)
      for (wk=0; wk<Wk; wk++)
       Ofm[n][co][ho][wo] +=
        Ifm[n][ci][ho+hk][wo+wk] *
        W[co][ci][hk][wk]
```

```
for (n=0; n<N; n++)
 for (co=0; co<Co; co++)
  for (ci=0; ci<Ci; ci++)
   par_for (hk=0; hk<Hk; hk++)
    par_for (wk=0; wk<Wk; wk++)
     for (ho=0; ho<Ho; ho++)
      for (wo=0; wo<Wo; wo++)
       Ofm[n][co][ho][wo] +=
        Ifm[n][ci][ho+hk][wo+wk] *
        W[co][ci][hk][wk]
```

FIGURE 34. Loop reordering of the 7-nested loops representation of the Conv layer.



$$\begin{bmatrix} I_0 & I_1 & I_2 \\ I_3 & I_4 & I_5 \\ I_6 & I_7 & I_8 \end{bmatrix} * \begin{bmatrix} W_0 & W_1 \\ W_2 & W_3 \end{bmatrix} = \begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix}$$

$P_0=I_0W_0+I_1W_1+I_3W_2+I_4W_3$ PE0

$P_1=I_1W_0+I_2W_1+I_4W_2+I_5W_3$ PE1

$P_2=I_3W_0+I_4W_1+I_6W_2+I_7W_3$ PE2

$P_3=I_4W_0+I_5W_1+I_7W_2+I_8W_3$ PE3

FIGURE 36. (left) Spatial mapping of the operations in an output stationary dataflow. (right) Modified PE for an output stationary accelerator.

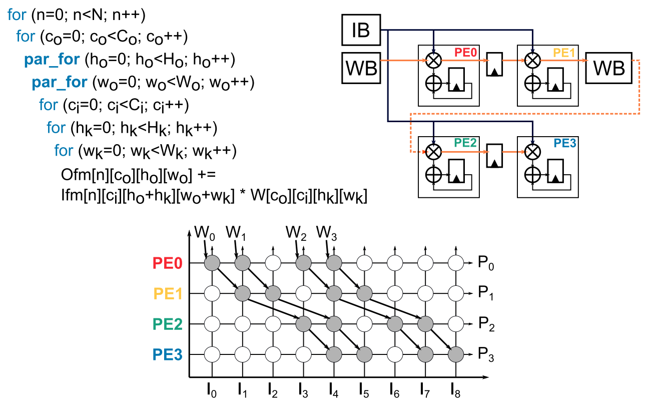
What characterizes the above-discussed dataflows is that all the operations along dimensions H_k and W_k are mapped to the 2D PE array and executed in parallel. This mapping operation is defined as *spatial unrolling* in [118]. From a software perspective this is equivalent to replacing the **for** loops in the 7-nested loop representation with parallel for loops (**par_for**) as in Figure 34. In [118], the $H_k|W_k$ syntax is adopted to denote which loops are parallelized. The stationarity of the weights is instead equivalent, from the software perspective, to a *loop reordering* operation of the **for** loops, as shown in Figure 34. Other architectures that adopt a $H_k|W_k$ weight stationary approach are [119], [120] and [121].

A different dataflow, but in which the weights are still stationary, is obtained by spatially unrolling the dimensions C_o and C_i ($C_o|C_i$). As shown in Figure 35, the operations that must be performed are equivalent to a vector-matrix multiplication. It can be realized in hardware with a 2D systolic array. In essence, the weights are internally stored in the PEs, the inputs are horizontally forwarded, and the partial sums are accumulated along the vertical dimension. An example of $C_o|C_i$-weight stationary dataflow can be found in in the Tensor Processing Unit (TPU) [122] developed at Google. TPUs are deployed in datacenters, and it has therefore been possible to obtain statistics and metrics on real-life applications. It has been observed that CNNs, on which the development of HW accelerators is focused, actually represent the 5% of all applications used in datacenters [122]. For this reason, Google designers decided to focus on the acceleration of FC layers, which are inherently vector-matrix operations and can, therefore, be directly mapped to the matrix-multiply unit that is the heart of the TPUs.

Because of its flexibility, the systolic array is often used in configurable architectures that must support various layer types [123], [124] [125]. This solution is also adopted for the acceleration of Capsule Networks [126], [127] [128], that consist of Conv layers, Conv layers of capsules and FC layers of capsules.

5) OUTPUT STATIONARY

The output stationary dataflow has the purpose of minimizing the data movement necessary to store and load the partial sums in the GLB. With the weight stationary dataflow of Figure 32, for example, the partial sum of a single output pixel must be stored and reloaded to/from the GLB $(H_k - 1) \times C_i$ times. In the output stationary dataflow, the PEs are modified to have the possibility of locally accumulating the results of the MACs that they perform (Figure 36). Each PE is therefore responsible for the computations necessary to obtain an output pixel, whose partial sums are accumulated in a single RF.



```
for (n=0; n<N; n++)
 for (co=0; co<Co; co++)
  par_for (ho=0; ho<Ho; ho++)
   par_for (wo=0; wo<Wo; wo++)
    for (ci=0; ci<Ci; ci++)
     for (hk=0; hk<Hk; hk++)
      for (wk=0; wk<Wk; wk++)
       Ofm[n][co][ho][wo] +=
       Ifm[n][ci][ho+hk][wo+wk] * W[co][ci][hk][wk]
```

FIGURE 37. (top left) Reordered loops for an output stationary dataflow . (top right) PEs array and (bottom) temporal-spatial mapping of the operations in an output stationary accelerator with input broadcasting and weight forwarding.

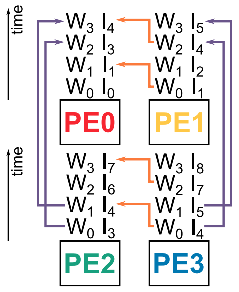


FIGURE 38. Data movements in ShiDianNao accelerator [129].

Similarly to the weight stationary dataflow, it is possible to spatially unroll the H_o and W_o loops to get an output stationary dataflow. The input pixels and the weights can then be read from the GLB and moved to the PEs array in different ways. It is, for example, possible to broadcast the input pixels to all the PEs and to forward the weights, as shown in Figure 37.

A popular accelerator that adopts an output stationary dataflow with $H_o|W_o$ spatial unrolling is ShiDianNao [129]. Being an output stationary dataflow, each PE in the 2D grid of ShiDianNao processes a pixel of an output feature map, and all the results are then collected and stored in the global buffer. A single weight is broadcasted to all the PEs at every operation cycle. The PEs can read the input pixel either from the GLB, from their right neighbor or their lower neighbor. The PEs have a RF for the partial sum accumulation and two FIFOs to store input pixels for inter-PEs communication. Figure 38 schematizes data movement in ShiDianNao for a 2×2 array of PEs.

Computing the output pixels in parallel along the dimensions H_o and W_o is not the only possible solution to get an output stationary dataflow. Origami accelerator [130], for example, spatially unrolls three loops (H_k, W_k and C_o) and computes all the pixels along the output channel C_o in parallel, dedicating an accumulator to each one. In a compromise between [129] and [130], in [131] the output pixels along

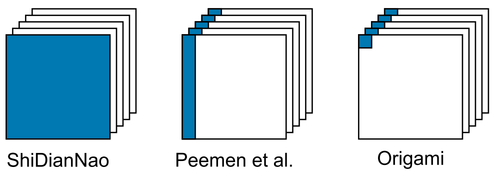


FIGURE 39. Different solutions [129], [131] [130] to spatially unroll the computations in an output stationary dataflow.

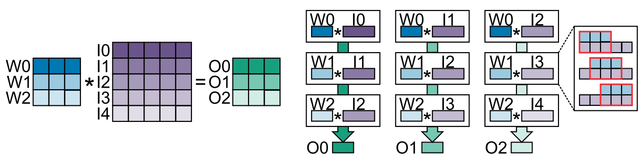


FIGURE 40. Operations mapping in a row stationary dataflow.

dimensions H_o and C_o are computed in parallel. Figure 39 graphically shows how [129], [131] and [130] spatially unroll the computation of the output pixels.

It is important to notice that spatially unrolling the dimensions C_i and C_o can either lead to a weight stationary or an output stationary dataflow. Beyond what data is kept stationary, $C_i|C_o$ dataflow is very common because it performs a vector-matrix or matrix-matrix multiplication, and therefore, it allows to easily map both a convolutional and a fully-connected layer to the same array of PEs.

6) ROW STATIONARY

The row stationary dataflow is introduced in [114] and used by the Eyeriss accelerator [132]. It has the purpose of maximizing the reuse of inputs, weights and partial sums all together, in contrast to weight and output stationary dataflows that focus on a single type of data reuse.

In the row stationary dataflow all the MACs necessary to perform a row of the convolution (1D convolution) are mapped to a single PE. A PE has a RF to keep stationary a row of the weight kernel while the inputs are streamed in the PE exploiting the sliding window mechanism. To perform a whole 2D convolution, it is necessary to have a 2D array of $H_k \times H_o$ PEs. Each column of the array computes the $H_k \times W_o$ partial sums that contribute to a row of the output feature map, that are therefore accumulated. Figure 40 shows how a 2D convolution with a 3×3 weight kernel is mapped to a row stationary dataflow, and how the partial sums are accumulated along the columns of the PEs array.

From Figure 40 it is also possible to see the different types of reuse obtained by the row stationary dataflow. The optimization of data reuse is multi-objective, i.e., a row of PEs shares the same weights, the input pixels are diagonally reused, and the partial sums are vertically accumulated.

7) NO LOCAL REUSE

The memory elements with higher energy efficiency are those with a low storage capacity, but they are less efficient in terms of area occupation (area/bit). Therefore, a RF has a

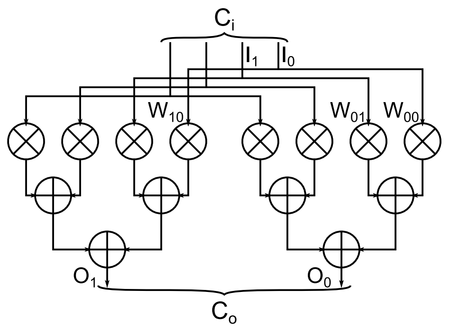
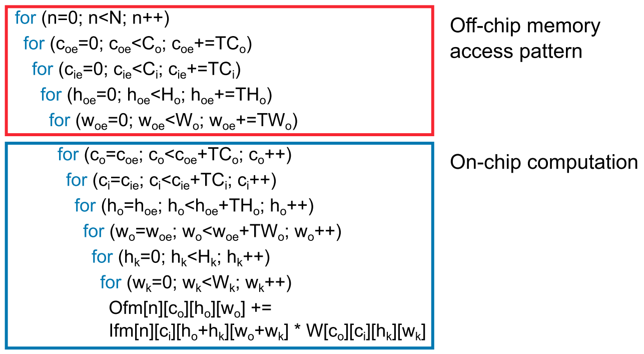


FIGURE 41. No local reuse dataflow.



```
for ($n=0; n<N; n++$)
  for ($c_{oe}=0; c_{oe}<C_o; c_{oe}+=TC_o$)
    for ($c_{ie}=0; c_{ie}<C_i; c_{ie}+=TC_i$)
      for ($h_{oe}=0; h_{oe}<H_o; h_{oe}+=TH_o$)
        for ($w_{oe}=0; w_{oe}<W_o; w_{oe}+=TW_o$)
          for ($c_o=c_{oe}; c_o<c_{oe}+TC_o; c_o++$)
            for ($c_i=c_{ie}; c_i<c_{ie}+TC_i; c_i++$)
              for ($h_o=h_{oe}; h_o<h_{oe}+TH_o; h_o++$)
                for ($w_o=w_{oe}; w_o<w_{oe}+TW_o; w_o++$)
                  for ($h_k=0; h_k<H_k; h_k++$)
                    for ($w_k=0; w_k<W_k; w_k++$)
                      $Ofm[n][c_o][h_o][w_o]$ +=
                      $Ifm[n][c_i][h_o+h_k][w_o+w_k]$ * $W[c_o][c_i][h_k][w_k]$
```

Off-chip memory access pattern
On-chip computation

FIGURE 42. Loop tiling technique applied to the 7-nested loops representation of the Conv layer. The outermost loops describe off-chip memory accesses, and the innermost loops determine the dataflow.

higher area/bit compared to a scratchpad memory or a SRAM. The no local reuse dataflow maximizes the area dedicated to storage by removing register files from the PEs and allocating all the on-chip memory in the global buffer. Having no local reuse in the PEs, the traffic from and to the GLB on the NoC will be higher.

Which dimension is spatially unrolled on the PEs is not relevant for the no local reuse dataflow. Two accelerators that adopt this dataflow are [24], [133] and [134], which execute the loops along the dimensions C_i and C_o in parallel. In [133], $C_i \times C_o$ multipliers are allocated to multiply the inputs and the weights, and the C_o outputs are then computed with adder trees. An input pixel is multicasted to C_o multipliers (see Figure 41), while each multiplier reads a different weight from the global buffer.

A critical aspect of the dataflow definition and accelerator design has not yet been mentioned. Usually, the global buffer size is not sufficient to fully contain the input feature maps, kernel weights and output feature maps. For this reason, it is necessary to apply the *loop tiling* technique, which consists of partitioning the larger tensors into smaller tensors that can be contained in the buffer. The **for** loops of the 7-nested loops representation of the convolutional layer are therefore split into multiple loops, as shown in Figure 42. The tiling factors (TC_o, TC_i, TH_o, TW_o) define the size of the innermost loops and consequently of the global buffer size. In contrast, the permutations of the outermost loops determine the off-chip memory accesses and how the data are reused.

Due to the wide variety of layer types and sizes in DNNs models, recently the reconfigurable accelerators that allow to efficiently map different types of layers on the same hardware have gained importance. For example, in [135], there are two 16×16 arrays, whose PEs are divided into general PEs and super PEs. The former are used to map the Conv and FC layers, while the latter are used for the activations functions, Pooling layers, and RNN layers. The arrays can also be partitioned to process multiple layers in parallel and the accelerator supports 8- or 16-bit operations. Another example of a reconfigurable accelerator is the NPU that is at the heart of Project Brainwave [136], the real-time AI FPGA used in Microsoft's servers. The Project Brainwave NPU is a spatially distributed architecture with efficient matrix-vectors multipliers for the operations between tensors and multifunction units that implement a wide variety of functions. MAERI [137] obtains reconfigurability through the interconnections. The multipliers are arranged in a 1D structure and the inputs are delivered with a flexible distribution network that can be set to implement different dataflows. Similarly, the outputs of the multipliers are collected with an Augmented Reduction Tree of adders. A similar approach is adopted in SIGMA [138], in which the flexible distribution and reduction networks allow to perform vector dot-products of different sizes simultaneously. Cerebras Wafer Scale Engine is the largest chip ever built, and it is optimized for DL applications. The engine consists of a large amount of flexible cores that target tensor operations but support general operations too. The memory has a high capacity, in the order of gigabytes, and is distributed on-chip. Huawei has released the DaVinci AI core [139], which is completely high-level programmable and consists of a vector engine and a 3D Cube engine for matrix computations. Two or more DaVinci cores can be combined to work in parallel, as in the Huawei Ascend 910 and 310 AI processors.

D. TOOLS FOR DESIGN SPACE EXPLORATION (DSE)

From the analysis of possible architectures and dataflows discussed in the previous section it can be understood that many aspects have to be considered during the design of an accelerator, such as, architectural parameters, memory hierarchy, spatial and temporal mapping, and tiling factors. Exploring the whole space of possible designs is a tough task (even an NP-hard problem considering a wide range of design points), especially if the target platform of the accelerator is an ASIC, whose development cost is high in terms of cost and time. For this reason, many researchers have been focusing on the development of tools and frameworks for efficient design space exploration (see Table 3).

Peemen *et al.* [131] proposed a design flow that selects the best computation schedules to maximize data reuse for a determined on-chip buffer size, exploiting loop reordering and tiling. The whole design space is explored to find the optimized schedule that minimized off-chip memory accesses,

TABLE 3. Comparison of the tools for Design Space Exploration.

Name	Year	Motivation	Target	Characteristics
Peemen et al. [131]	2013	Minimize off-chip memory accesses	MEM	DSE of all schedules, given on-chip buffers sizes
Pouchet et al. [140]	2013	Minimize off-chip memory accesses	MEM	Local memory promotion
Yang et al. [141]	2016	Optimize scheduling given multi-level memory hierarchy	MEM	Systematic approach (i.e., iterative optimization) to loop blocking
SmartShuttle [142]	2018	Minimize off-chip memory accesses	MEM	Adapt tiling factors and data reuse to the size of the convolution
NNest [143]	2018	Optimize memory hierarchy, memory accesses and computational resources (CR)	MEM + CR	Given the tiling factors, optimization of the computational resources, of the on-chip buffers and of the schedule
RomaNet [144]	2019	Minimize off-chip memory accesses	MEM	Given a NN layer, optimization of tiling and partitioning
MAESTRO [145]	2019	Analytical cost model used for DSE	CR	Given a DNN, a HW configurations and a schedule, estimation of energy, execution time and NoC
mRNA [146]	2019	Adapt scheduling to computational resources	CR	Optimize the scheduling targeting MAERI accelerator and its resources
Timeloop [147]	2019	Data mapping exploration and evaluation	MEM + CR	Given an hardware accelerator, explore all the possible data mappings and evaluate them
MAGNet [148]	2019	Hardware and mapping generation and optimization	MEM + CR	Given HW and performances constraint, generate a HW accelerator and a valid mapping; use DSE to co-optimize HW and SW
XploreDL [149]	2020	DSE of hardware configurations	MEM + CR	Accurate HW simulation and evaluation to perform the DSE
SuperSlash [150]	2020	Minimize off-chip memory accesses	MEM	Integration of pruning with existing DSE techniques

discarding the configurations that do not satisfy the memory size requirement.

In [133], loop tiling is realized so that the innermost loops represent on-chip computation and the outermost loops the off-chip memory accesses, as in Figure 42. Local memory promotion [140] is then used to eliminate redundant memory accesses. If one among the input feature maps, output feature maps or weights is not addressed by the index of the innermost off-chip loop (w_{oe} in Figure 42), then it is reused for all its iterations. Hence, there is no need for continuously loading and storing back the reused tensor. The operations of load and store can consequently be moved out of the innermost loop. A polyhedral-based optimization is used to identify all the possible combinations of loop schedules and tiling factors, and local memory promotion is applied whenever possible. The computational roof and the computation to communication ratio is calculated for each solution to identify the optimal one.

In [141], Yang *et al.* showed a systematic approach to loop blocking. Given a memory hierarchy, the systematic approach consists of applying a *loop blocking* (i.e., loop tiling and loop reordering) for each level of the memory hierarchy. Exploring the design space for a multi-level memory hierarchy with the proposed methodology is computationally expensive. Therefore, Yang *et al.* proposed an iterative optimization where the loop blocking is applied to two levels of memory at a time, starting from the lowest level to the highest and re-adjusting the lower levels parameters at each iteration.

SmartShuttle [142] is a framework that focuses on optimizing off-chip memory accesses exploring the possible loop schedules, that influence the data reuse, and the tiling factors. In [142], it is noted that convolutional layers with different shapes may benefit from different types of data reuse and various tiling factors. SmartShuttle therefore adaptively varies the ordering and tiling of the loops to match different convolutional layers dynamically.

NNest [143] is a design space exploration tool for inference accelerators that focuses on the optimization of the memory hierarchy, of the memory accesses and the computational resources too, covering all the main aspects of an accelerator design. In [143] it is proposed a spatial accelerator architecture template that is parametrized, with the possibility of setting the tiling factors, that directly define the size of the on-chip buffers, the size of the PEs array and the possibility of implementing different dataflows and reuse schemes. NNest explores the whole design space and finds the Pareto-optimal solutions for a NN layer. It also allows for a multi-layer fitting.

In ROMANet [144], a systematic design space exploration methodology is proposed for reducing the number of memory accesses required for DNN inference. For each layer, an efficient data partitioning and scheduling is designed, based on the available on-chip memory and data reuse factors. Moreover, the proposed DRAM data mapping reduces the number of DRAM row buffer conflicts, while improving the system throughput, compared to a conventional DRAM design.

MAESTRO (Modeling Accelerator Efficiency via Spatio-Temporal Reuse and Occupancy) [145] is an analytical cost model that estimates the execution time, energy and NoC of a hardware configuration applied to a DNN model with a specific dataflow. MAESTRO is a cost model precise and efficient enough to be used for design space exploration, and can be used to determine Pareto-optimal architectural parameters given area, energy or throughput constraints.

mRNA [146] is a mapper that performs design space exploration to find the optimal mapping targeting the reconfigurable DNN accelerator MAERI [137]. Similarly to other design space exploration tools, it explores all the possible permutations of the **for** loops of the Conv layer 7-nested loop representation and all the possible combinations of tiling factors. Given the high dimensionality of the design space, mRNA reduces it by applying constraints based on domain knowledge, for example setting the tile sizes as multiples of the number of multipliers contained in MAERI to maximize resource utilization. mRNA experiments confirm that dataflows that maximize the usage of available PEs have a shorter runtime and that exploiting data reuse and broadcast/multicast reduces the energy cost.

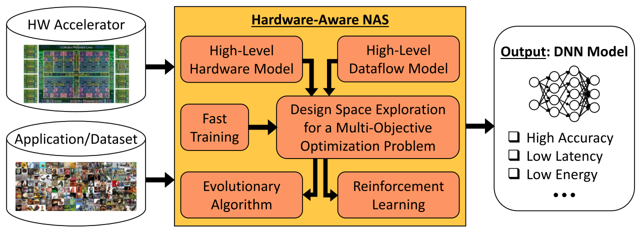


FIGURE 43. An overview of the hardware-aware NAS problem.

Timeloop [147] is a framework for the exploration of the design space of DNN hardware accelerators and for the evaluation of their performance and energy consumption to make the design more systematic. The users can describe an architectural model following a configurable template and, given a workload, a mapper within Timeloop systematically constructs the map space to be explored and evaluates every possible mapping with its performance, area and cost models.

MAGNet [148] is a Modular Accelerator Generator for Neural Networks that consists of the following three modules. (1) A MAGNet Designer, that, given a neural network model, hardware constraints and performance goals, generates an accelerator based on a parametric template. (2) A MAGNet mapper that generates a valid mapping of the operations on the accelerator, defining the tiling factors, the spatial and temporal mapping. (3) A MAGNet tuner that uses Bayesian optimization to efficiently explore the design space for hardware-software co-optimizations.

XploreDL [149] is a design space exploration tool for both training and inference accelerators. The tool can be employed in an early stage of the design, because it estimates in a fast yet fidelitous way the Pareto-optimal solutions for specialized accelerators executing CNNs and Capsule Networks, given as optimization objectives the energy-efficiency and the performance-per-area.

Since different level of DNN compression show different on-chip memory accesses, depending upon the pruning strategy, SuperSlash [150] integrates the pruning techniques with existing design space exploration methodologies, evaluating multiple data reuse strategies for each layer. For instance, the off-chip memory access volume can be reduced by directly using a layer's output as the input for the processing of the subsequent layer.

E. HARDWARE-AWARE NEURAL ARCHITECTURE SEARCH

Another interesting design strategy is to customize the DNN based on the underlying hardware. The optimization goal is then to jointly optimize the accuracy and the energy-efficiency, given the underlying hardware (e.g., an accelerator) and the dataset for the target application, as shown in Figure 43.

One of the biggest challenges is caused by the explosion of the exploration time and space, when all the hyper-parameters of the DNN are considered. To overcome such a problem, a fast yet accurate evaluation of the energy consumption and performance of the hardware is key. Therefore, a high-level modeling of the scheduling and dataflow, as discussed in the previous sections, is required. Moreover, a smart search is typically employed to speedup the exploration convergence. In the literature, there exist mainly three types of heuristic search algorithms for the hardware-aware neural architecture search, which are (1) evolutionary algorithms, (2) reinforcement learning, and (3) differentiable NAS.

The ProxylessNAS [151] can reduce the computational demand of the search by executing partial tasks, such as training on a smaller dataset, or learning with only a few blocks, or training just for a few epochs. Afterwards, the framework can directly learn the architectures for the complete tasks and the target hardware platforms. The MnasNet approach [152] directly implements and measures the inference latency by executing the model on mobile phones, and incorporates the model latency into the main objective of the search, along with the accuracy. In [153], the authors proposed a black-box profiling-based search in the first stage of the accelerator-aware NAS pipeline using an ISA-based DNN accelerator on FPGA, with a particular focus on the accurate latency evaluation. The NASCaps [154] is a framework integrating capsule layers in the search space. With a multi-objective evolutionary algorithm, it jointly optimizes the accuracy and the hardware efficiency of capsule-based DNNs. In [155], the authors developed a NAS framework which integrates the quantization and hardware implementation in the design flow.

The APNAS [156] is a reinforcement learning-based exploration methodology, searching for high accurate DNNs that also offer high execution performance. To speed-up the search, instead of running millions of DNN configurations on real hardware, the cycle count is estimated by analytical models. The FNAS framework [157] performs a hardware-aware NAS targeting FPGA acceleration. In particular, it employs an abstraction model to estimate the latency for meeting the specifications. Moreover, a specialized scheduling mechanism is proposed to execute the DNN inference on multiple FPGAs. The HotNAS [158] is a hardware and neural architecture co-search methodology, which starts the exploration from a set of existing pre-trained models to reduce the training time. In addition, it supports hardware for compressed DNNs and it integrates the compression in the co-search to improve the energy-efficiency. With the ENAS approach [159], the authors proposed to share the parameters between the child DNN models. It allows not only to speedup the search, but also to achieve high accuracy, with similar benefits as for the transfer learning [160].

The Single-Path NAS [161] is a method searching for the optimal building block for the convolutional layers, called superkernel, and then sharing the convolutional kernel weights with a specialized encoding. The DNAS [162] is a differentiable NAS framework, where the search space is represented by a stochastic super net. It explores a layer-wise space where each layer of the CNN can choose a different block, and the learning process is done by training the

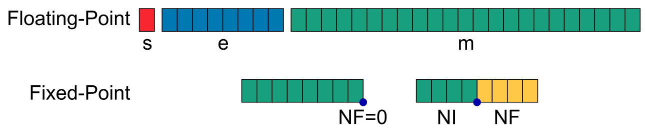


FIGURE 44. (top) 32-bit floating-point representation. (bottom) 8-bit dynamic fixed-point representation.

stochastic super net. The SPOS [163] uses in a similar way the supernet concept to perform NAS, where the constraints such as latency and number of FLOPs are applied. The HURRICANE framework [164] performs a two-stage search algorithm for the automatic hardware-aware NAS. It can generate different models for different types of hardware platforms for executing the inference. In [165], the authors demonstrate that competitive results for the NAS can be achieved by using random search. This approach significantly reduces the complexity, compared to other search methods.

F. FULL PRECISION VS QUANTIZED IMPLEMENTATIONS

As discussed in the previous sections, one of the main obstacles to the deployment of DNNs on edge devices is their large memory footprint, the high energy cost of memory accesses, and the energy required for computations.

One of the most popular methods for reducing memory and computation requirements is quantization. Quantization is the process of mapping values from a continuous or large set to a discrete and smaller set by applying a function that can be either linear or non-linear. The difference between the quantized value x_q and the original value x is the quantization error e_q (Eq. 6).

$$e_q = x_q - x \tag{6}$$

From a hardware perspective, quantization reduces the precision of the values, and consequently, the number of bits necessary to represent them. It is, therefore, possible to move from the floating-point representation to a shorter fixed-point representation (see Figure 44). According with the IEEE 754 standard, in 32-bit floating-point representation, one bit expresses the sign s of the number, 8 bits represent the exponent e and 23 bits the mantissa m. The value of the number is $(-1)^s \cdot m \cdot 2^{e-127}$ and can be in the range of 10^{-38} to 10^{38}. An N-bit fixed-point number in two's complement has an integer part of NI bits and a fractional part of NF bits. The width NF of the fractional part can be seen as a *scale factor* that determines the position of the decimal point. Numbers can be in the range $[-2^{NI-1}, 2^{NI-1} - 2^{-NF}]$ and the quantization step is 2^{-NF}.

The scale factor NF can be varied to have different ranges and different precision, making the fixed-point representation *dynamic*. This is particularly useful for neural networks, as weights and activations fall in very different numerical ranges depending on the layer.

The fixed-point representation allows for memory and energy saving, e.g., a MAC performed on an 8-bit fixed-point number consumes 20x lower energy than a MAC on a 32-bit floating-point number [166]. Moreover, a number expressed on 8 bits has a memory footprint of 4x smaller than one on 32 bits. This allows us to understand the large potential of gain, in terms of energy and memory, that can be achieved through quantization of data.

The purpose of quantizing the neural networks is to reduce the size of the models, obtaining a lower memory footprint and at the same time, a lower energy cost for both the computations and memory accesses. However, quantization carefully must be applied without reducing the accuracy of the models.

In NNs, there are three sets of values that can be quantized: the weights, the activations and the gradients. Earlier works on quantized NNs focused on the weights only since they directly affect the memory requirements [167], [168]. While the activations must be quantized at each execution of the algorithm, the weights can only be quantized once off-line after the training. This has two advantages:

- The quantized weights can be further fine-tuned to recover a possible accuracy degradation following the precision reduction.
- Since the weights are quantized offline, it is possible to apply complex quantization functions or stochastic functions, without affecting the computational resource required on-chip.

Recently researchers have started to focus on the quantization of activations too [169], [170] [171], that affect the memory footprint and bandwidth depending on how the dataflow is implemented, as well as directly affecting the required computational resources.

The study of gradient quantization is limited, mainly for two reasons:

- The training of a NN is very sensitive to even small variations in weight values, and there is a risk of not achieving convergence. Therefore quantizing the weights is complex.
- Usually the training is done only once offline on a GPU or a CPU, and not on the edge devices, so there is no reason to devote too much effort to reduce the size of the model and to reduce the energy consumption.

Several studies have been made on quantization methods [172]. In the following, we will provide an overview of hardware-friendly quantization methods, distinguishing between linear and non-linear methods (see Figure 45).

1) LINEAR QUANTIZATION

It is characterized by evenly-spaced quantization intervals, as shown in Figure 45.a. An example of linear quantization is the above-discussed fixed-point coding, which has been widely studied and applied to NNs because its hardware implementation is well known. Moreover, most CPUs support fixed-point arithmetic on 8, 16, or 32 bits. Given the strong diffusion that the quantization of NNs is having, the Nvidia Tesla GPU supports 8-bit fixed-point operations, and so do

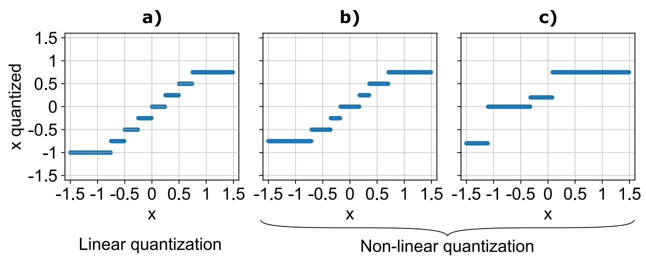


FIGURE 45. Linear, logarithmic and vector quantization techniques.

TABLE 4. Comparison of different variable-bitwidth HW platforms.

Name	Weights	Activations	Features	Target
BISMO [178]				
Stripes [179]	16-bit	1-bit to 16-bit	Serial	ASIC
UNPU [180]	1-bit to 16-bit	16-bit	Serial	ASIC
Loom [181]	1-bit to 16-bit	1-bit to 16-bit	Serial	ASIC
Bit Fusion [182]	1,2,4,8,16-bit	1,2,4,8,16-bit	Spatial	ASIC
BitBlade [183]	1,2,4,8,16-bit	1,2,4,8,16-bit	Spatial	ASIC
Turing TC [184]	64,32,16,8,4-bit	64,32,16,8,4-bit		GPU
PowerVR S2NX	16,8,4-bit	16,8,4-bit		SoC

the Tensor Processing Units (TPUs) [122] used in Google datacenters.

It has been demonstrated by several works that both the weights and activations can be quantized to 8-bit dynamic fixed-point numbers for inference without significantly affecting the accuracy [173] [169]. The Ristretto framework [173] identifies the quantization parameters (bitwidth and scale factor) by running a statistical analysis on the weights and activations. The weights are furtherly fine-tuned with a re-training step. With the Ristretto framework, complex models such as AlexNet [40], SqueezeNet [174] or GoogleNet [43] are inferred on 8 bits with less than 1% accuracy loss. In [170], an NN for speech recognition is implemented on 8-bit fixed-point numbers exploiting the Intel SSSE3 instruction set for SIMD execution. A speed-up of 7.6x is achieved compared to the floating-point baseline.

Given the great diversity between the various layers of an NN, it may be useful to use a different precision across the model, i.e., a variable bitwidth depending on the layer. Works in [117] and [175] show that the bitwidth can be set depending on the position in the model, making a per-layer optimization of the number of bits of weights and activations. In particular, in [175], it is stated that the bitwidth used for the weights can decrease approaching the last layers of the NN, while the bitwidth of the activations remains more or less constant. Following these ideas, Q-CapsNets [176] analyzes the layer-wise quantization capabilities of weights and activations of CapsNets, with a cross-layer optimization of the bitwidth and a fine-grained tuning for the dynamic routing operations. Finding the optimal bitwidth for each layer of a DNN is a complex task. For this purpose, HAQ, a hardware-aware quantization framework, is introduced in [177]. It applies reinforcement learning to determine the optimal bitwidths for weights and activations, using as feedback the results of a hardware simulator.

The research on fine-grained bitwidth optimization is also backed by the parallel development of hardware accelerators that support flexible bitwidth arithmetic operations. BISMO [178] is a matrix-matrix multiplication core with variable parallelism and precision to adapt to the requirements of different applications. It supports precision from 1 to 8 bits exploiting bit-serial computation. Stripes [179] is an accelerator for DNNs with flexible bitwidth for the activations that uses bit-serial operations. UNPU [180] has a similar approach, but the bits of the activations are kept constant to 16-bits and the weights have variable bitwidth. Loom [181] adopts bit-serial multiplicators and both weights and activations have fully variable bitwidth, from 1-bit to 16-bit. Bit Fusion [182] instead implements variable precision operations for DNNs with a spatial approach, using an array of bit-level PEs combined together according to the required bitwidth. BitBlade [183] is an optimization of Bitfusion, in which bit-wise summations substitutes the shift-add logic. On the industrial front, in 2018 Apple released the A12 Bionic chip with a Neural Processing Unit (NPU) that supports variable precision; Nvidia Turing Tensor Cores, available in the Nvidia Turing architecture [184], support operations from 32/16-bit floating-point down to 8/4-bit fixed-point; the Imagination PowerVR Series2NX architecture has adjustable bitwidth from 16 to 4 bits. The above-discussed platforms that provide variable bitwidths are compared in Table 4.

Both weights and activations can be quantized to very low bitwidths. BinaryConnect (BC) [167] introduced the idea of binary weights, included in $\{-1, 1\}$. Binary Weight Nets (BWN) [185] approximate a filter W as αB, where B is a filter whose values are binary, and α is a scale factor. The operations are performed between full-precision inputs and binary weights, and the output is then multiplied by α. In Ternary Weight Nets (TWN) [168] the same approach is adopted but the weights are ternary, i.e., in the set $\{-1, 0, 1\}$.

Quantized Neural Networks [171] and DoReFa-Net [186] have an even more aggressive approach using binary weights and 2-bit activations. Finally, Binarized Neural Networks (BNN) [187] and XNOR-Nets [185] use both binary weights and activations. XNOR-Nets use the same approach as BWN and TWN to limit accuracy reduction by multiplying the outputs with a scaling factor.

Several hardware accelerators have been proposed to support efficient inference of binary NNs: YodaNN [188] and Hyperdrive [189] are accelerators for binary weights only NNs; BRein [190], XNOR Neural Engine [191] and XNORBIN [192] accelerate NNs with binary weights and activations, while BRein supports ternary weights too.

2) NON-LINEAR QUANTIZATION

Weights and activations in an NN usually have non-uniform distributions, so they can benefit from the non-linear quantization, where the quantization intervals are unevenly distributed, as shown in Figure 45.b and 45.c.

An example of a non-linear quantization scheme is the logarithmic quantization, first applied to NNs in [193]. The dot product between a vector of weights w and activations x can be approximated as follows adopting the logarithmic quantization:

$$w \cdot x = \sum_{i=0}^{N} w_i x_i \simeq \sum_{i=0}^{N} w_i 2^{\tilde{x}_i} = \sum_{i=0}^{N} w_i \ll \tilde{x}_i \tag{7}$$

$$\tilde{x}_i = Int(log_2(x_i)) \tag{8}$$

From Eq. 7 and Eq. 8, we can notice that the multiplications can be substituted with shift operations. With the same bitwidth used, logarithmic quantization reduces the accuracy loss compared to linear quantization. With respect to a floating-point baseline, the accuracy loss of VGG16 with a 3-bit linear quantization is 6.2%, while with logarithmic quantization it is only 0.6%.

In [194], [195] and [196], NN accelerators with logarithmic numerical representation are presented. They are characterized by processing elements whose multipliers used for MACs are replaced by barrel-shifters.

Another type of non-linear quantization is *vector quantization*. It consists of applying clustering algorithms to the weights of an NN. The centroids of the clusters to which the weights belong are used as quantization values. For the first time, this method was applied to the quantization of NNs in [197]. The vector quantization can be applied offline before inference, so it does not need accelerators with specialized architectures to support it.

G. METHODS FOR MODEL COMPRESSION

As seen in Section II-C, the trend to achieve greater accuracy has been the development of deeper and deeper NNs with a higher number of layers and parameters. This evolution is hardly compatible with the recent desire to deploy NNs on mobile and edge devices. During the last few years, therefore, there has been a big push towards the research of methods to compress the models of NNs without affecting the achieved accuracy [198]. The most prominent works are in the domains of network pruning, architectural choices and knowledge distillation, as described in the following paragraphs.

1) NETWORK PRUNING

Given the redundancy of the parameters in NNs, *network pruning* consists of removing, i.e., set to zero, those parameters that do not affect the performance (i.e., network accuracy) of the model. Pruning was first explored in Optimal Brain Damage [199], where the weight with lower influence on the loss function during the training were pruned. A simpler method [200] consists of pruning the weights with small magnitude after the training and then in performing a fine-tuning of the remaining weights to recover possible accuracy losses. This method, straightforward and linear, allows to reduce the number of parameters in AlexNet, for example, by 10x [200].

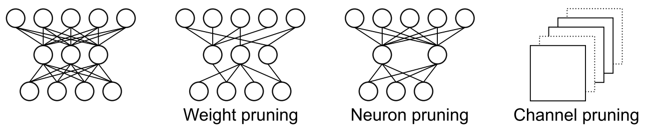


FIGURE 46. Various pruning techniques.

Subsequent works have proposed variations of the pruning method in an attempt to obtain a high yet accuracy-wise effective compression of the models. In [201], instead of removing individual weights, entire neurons are pruned. In [202], full channels are pruned from feature maps by applying a two-step algorithm based on LOSSA regression for channel selection and least square reconstruction. In [203], Deep Compression is proposed, a three-stage pipeline that applies, in order, pruning, quantization and Huffman coding. PruNet [204] iteratively applies a magnitude-based Class-Blind pruning followed by weight retraining. In [205], the pruning is guided by an estimate of CNN's energy consumption, to optimize the model's energy performance and not just minimize the number of parameters. A similar approach based on energy constraints is adopted in ECC [206]. In [207], quantization and pruning are performed jointly, and fine-tuning is run in parallel. In AMC [208] and [209], learning-based approaches are adopted to prune and quantize the models for algorithm-hardware co-design. In APQ [210], pruning and quantization are optimized jointly with the NN model avoiding any accuracy loss.

The pruning has the advantage of making the matrices of the weights sparse. Section III-H explains in detail how it is possible to take advantage of *sparsity* in neural networks.

2) ARCHITECTURAL CHOICES

Some researches have explored new architectures with a lower number of parameters by construction. The basic idea is to replace a large kernel with a series of two or more smaller kernels. In this way, an equivalent receptive field is obtained but with fewer parameters. For example, a 5×5 kernel can be replaced by a series of two 3×3 kernels, reducing the number of weights from 25 to 18 (see Figure 47). In SqueezeNet [174], most of the 3×3 kernels are substituted with 1×1 kernels that have 9x fewer parameters, and the input channels to the 3×3 convolutions are reduced. SqueezeNet achieves the same accuracy of AlexNet with 50x fewer parameters. In MobileNet [211], a standard convolution is divided in a depthwise convolution and a point-wise convolution. The depthwise convolution applies a different kernel to each input channel, while the point-wise convolution uses 1×1 kernels to combine together the output channels of the depthwise convolution. This factorization reduces the number of parameters. Xception [212] adopts this same approach.

It is also possible to obtain smaller tensors from large tensors after training by applying the Tensor Decomposition,

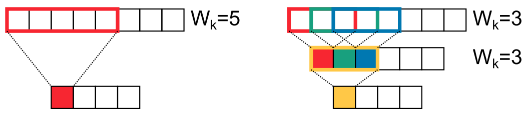


FIGURE 47. Reduction of the number of parameters by splitting a large kernel in a series of two smaller kernels while maintaining the equivalent receptive field equal.

which is a low-rank factorization technique. The kernels of the convolutional layers are 4D tensors, while the weights of the fully-connected layers are organized in a 2D matrix. With tensor decomposition, these can be broken down into tensors of lower dimensionality by Canonical Polyadic (CP) decomposition [213]. Since CP is not numerically stable for tensors with dimension higher than two, it is possible to adopt Tucker decomposition [214].

3) KNOWLEDGE DISTILLATION

Higher accuracies are obtained with very deep models or with *ensembles* of models, whose results are then averaged. Using a deep model or even several models at once requires considerable computational effort. However, it is possible to transfer the knowledge of one or more large models (teachers) into a smaller model (student). This process is commonly known as *knowledge distillation* and has been introduced in [215] and [216], for shallow and deep teacher models respectively. In [215] and [216], the (trained) teacher models receive a dataset of unlabeled data and classify them, producing a synthetically-labelled dataset. This dataset is then used to train the shallow student model, that, therefore, learns to mimic the classifying function of the teachers. The knowledge distillation method has shown promising results and several variations have been proposed in subsequent work [217], [218] [219], [220].

H. ACTIVATIONS AND WEIGHTS SPARSITY: STRATEGIES AND ENCODING

Recent studies have shown that most DNNs are subject to redundancy concerning the weights. Consequently, it is possible to prune them without affecting the accuracy as demonstrated in [200] and [201]. Both works show that the synapses can be reduced to percentages ranging from 20% to 80%, depending on the considered layers. As explained in Section III-G, pruning weights results in zero values that make the matrices sparse. On the other hand, during inference, the ReLU clamps negative activations to zero. The null activations range from about 50% to 70%. Hence, these represent the two primary sources for sparse matrices for both activations and weights. Sparsity represents an excellent opportunity to optimize the inference for two main reasons:

- The basic operations of the DL is the multiplication between a weight and an activation. However, whenever one of the two is zero, the operation has no reason to be performed, as the result, will be null too. Therefore, it is possible to skip such operations to speed up execution and save the energy.
- By using compression techniques, it is possible to save only the non-null elements and their relative positions in the matrices. This reduces the storage requirements with the possibility of fitting more data into the on-chip SRAM, thereby cutting off the accesses to the off-chip DRAM significantly.

Although numerous compression techniques can be found in literature, DNNs and CNNs rely mainly on three hardware-friendly methods. Such methods consist of two sets of data: one represents all the non-zero values, while the other represents the metadata or the indices necessary to reconstruct the original pattern. Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) are two formats belonging to the class of compressed stripe storage [221]. Both CSR and CSC can be seen as a collection of scattered vectors, which allows random access to entire rows or columns respectively, equipped with an efficient count of non-zeros within each row or column, as detailed in the following.

1) COMPRESSED SPARSE ROW (CSR)

As shown in Figure 48(a), a single array (Non-zero array) stores all the non-zero values of the sparse rows in order, and an integer array (Column indices) stores the corresponding column indices. A third array (Row pointer) stores the offsets within the previous two vectors, indicating the number of non-null elements per row in an incremental fashion. Such a structure allows fetching any row thanks to an efficient element enumeration. The number of bits required for such a representation is given as:

$$I \cdot (Nb \cdot (1 - Sp) + Ni \cdot (1 - Sp)) + (H + 1) \cdot No \quad (9)$$

where I is the input size, Sp the sparsity percentage, H the height of the input matrix, and Nb, Ni and No are the bitwidths of data, indices and offset, respectively.

2) COMPRESSED SPARSE COLUMN (CSC)

CSC works like CSR, but this time data are organized by columns. A single array (Non-zero array) stores all the non-zero values of the sparse columns in order, and an integer array (Row indices) stores the corresponding row indices. A third array (Column pointer) stores the offset within the previous two vectors, indicating the number of non-null elements per column in an incremental fashion. Figure 48(b) shows an example of the CSC coding. The number of bits required for such a representation is quite similar to the previous one, i.e.,:

$$I \cdot (Nb \cdot (1 - Sp) + Ni \cdot (1 - Sp)) + (W + 1) \cdot No \quad (10)$$

where W is the width of the input matrix.

3) COMPRESSED IMAGE SIZE (CIS)

This data format consists of a sparsity map and a non-zero value list, as depicted in Figure 48(c). The former is a mask

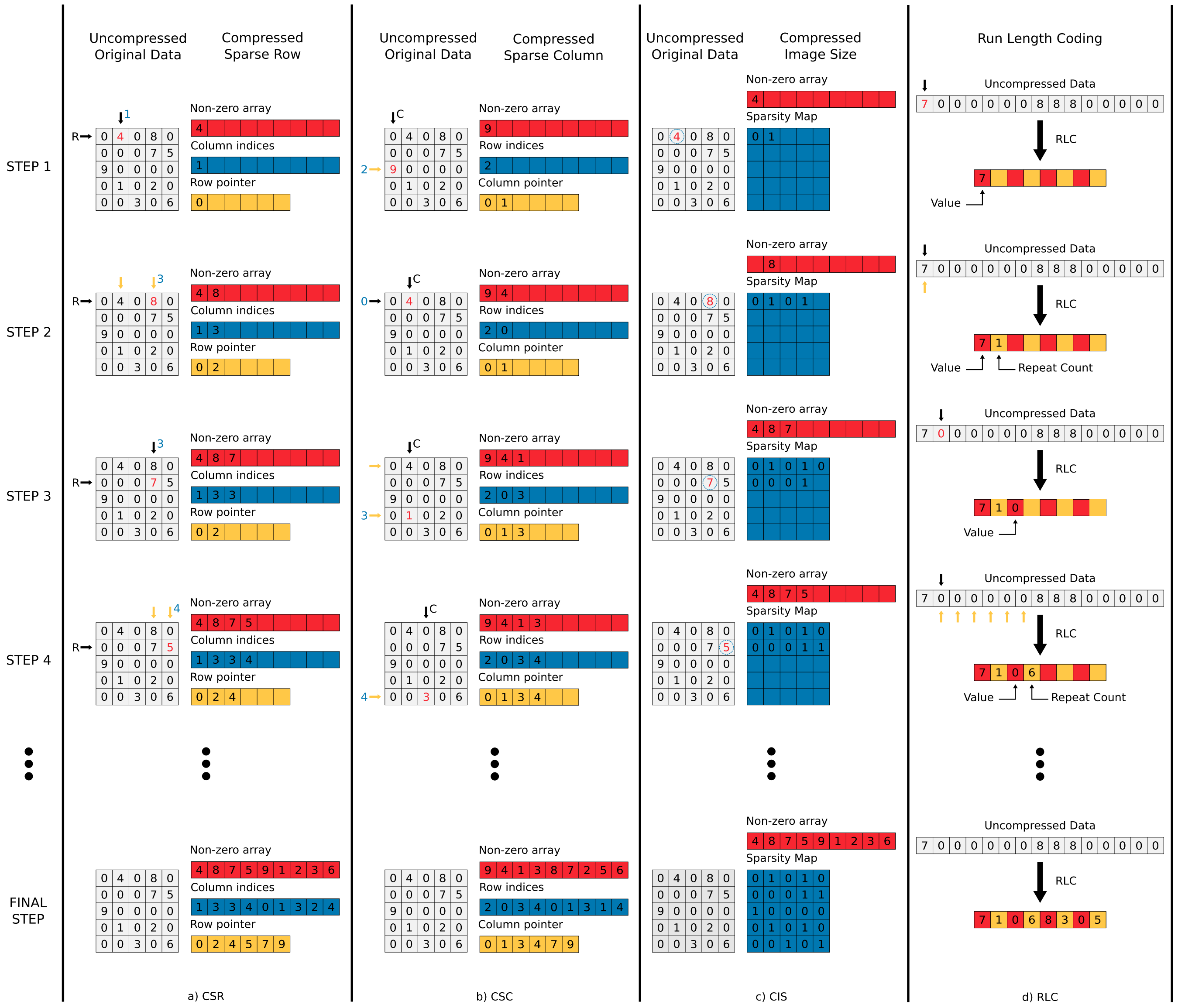


FIGURE 48. Step-by-step compression formats comparison: (a) Compressed Spare Row (CSR), (b) Compressed Sparse Column (CSC), (c) Compressed Image Size (CIS), (d) Run Length Coding (RLC).

with the same shape of the original data (1D vector, 2D Matrix or 3D matrices array) having one bit per entry. The bit is 0 if the corresponding value is null, 1 otherwise. The latter is an array composed of all non-zero values. With respect to CSR and CSC this technique allows an easier representation with no need for decompression. In this case, the number of required bits has a simpler equation, as given below:

$$I \cdot (Nb \cdot (1 - Sp) + n) \tag{11}$$

where n is typically 1 bit.

Figure 49 compares the compression ratio of CSR, CSC and CIS methods. This is the ratio between the compressed bit size and the original model. The picture includes two boundaries for the three formats. The upper case is based on the filter size of Conv 4 for AlexNet ($3 \times 3 \times 384$) with data parallelism of 8 bits, while the lower bound is based on Conv 1 for the same neural network ($11 \times 11 \times 3$) represented on 32 bits. As it is possible to notice, the CIS format performs better than the CSR or CSC formats in almost all the sparsity range. However, the coding choice often depends on how the data will be handled by the hardware.

For the sake of completeness, a fourth method should be introduced, namely Run Length Coding (RLC).

4) RUN LENGTH CODING (RLC)

It is a simple data format that is able to compress the consecutive repetition of the same value as depicted in Figure 48(d). In the case of sparsity, it is mainly used to compress consecutive zeros in a single zero and the count of them. Although it is very easy to implement, it is only effective when zeros

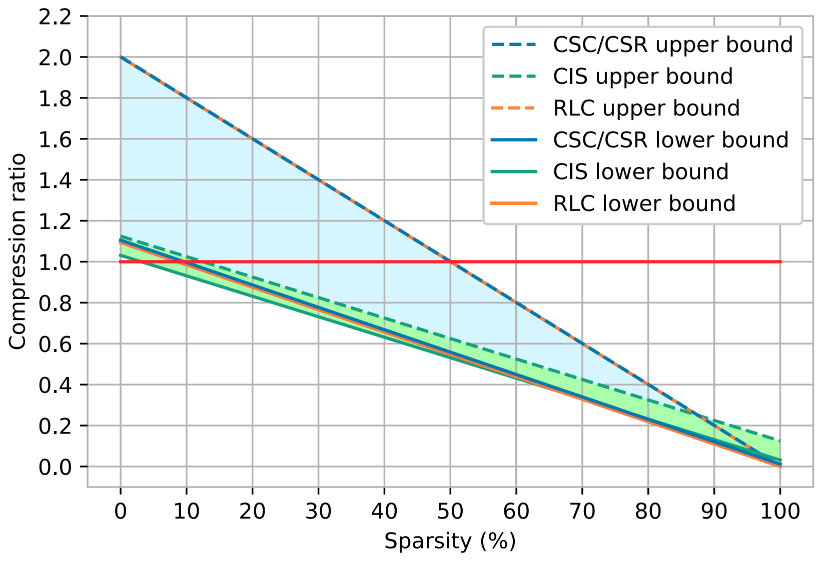


FIGURE 49. Compressed ratio with variation in sparsity of CSC, CSR, CIS and RLC. The picture includes two boundaries: the upper bound refers to Conv 4 of AlexNet (3 × 3 × 384) with data parallelism of 8 bits, while the lower bound is based on Conv 1 (11 × 11 × 3) represented on 32 bits.

manifest in a compact and consecutive manner (high percentage of sparsity). In addition, the RLC is designed for data arrays, so it is not optimal when operating on matrices.

The following, and last proposed method represents a milestone in the history of compression. However, for reasons of complexity it is difficult to use in hardware architectures.

5) HUFFMAN CODING

As is well known, Huffam coding is the most efficient method to encode scattered data thanks to its optimal compression rate. However, its complexity makes it difficult to employ since it would require computation-hungry compressor/decompressor schemes (large silicon area). Moreover, the continuous data manipulation would introduce a power overhead, which can hardly be compensated by saved computations. Thus, such a non-friendly-hardware coding approach is only used in software-level implementations.

Even though it is proved that the above-mentioned techniques bring benefits, compression introduces irregular data patterns that are reflected in irregular memory accesses. Moreover, ad-hoc hardware support is required to identify useful operations. In this scenario, general-purpose platforms like CPUs and GPUs are not very prone to use sparsity as an advantage, but rather, random memory accesses represent for those a source of inefficiency.

Thus, many FPGA and ASIC architectures leverage sparse matrices to accelerate the inference stage thanks to custom hardware. For example, Cnvlutin [222] relies on the ReLU function to compress activations with a CSR approach, but it does not consider weight sparsity. On the other hand, Cambricon-X [113] employs the weight sparsity, having a PE-based implementation, where each PE stores compressed synapses for asynchronous computation. SCNN [223], instead, is able to take care of both sparsity simultaneously in CNNs by means of an input stationary dataflow. Activations and weights in a CSC scheme are provided to a multiplier array that generates scattered partial products, subsequently added together using a dedicated interconnection mesh. Despite the fact that it reaches an excellent PEs utilization efficiency over convolutional layers, fully connected ones represent a bottleneck since it is impossible to reuse values. EIE [224] encodes the sparse weights using the CSC format as well, avoiding the use of the DRAM for 120x energy saving. Moreover, the ability to skip zero activations makes its matrix-vector multiplication inference engine extremely efficient. NullHop [225] is a CNN FPGA-based hardware accelerator which embodies both a zero-skipping ability over null activations and a CIS compression over the synapses. The first comes without any clock cycle waste, while the second allows acting directly on compressed data thanks to its hardware-friendly representation. SqueezeFlow [226] exploits a different approach by introducing concise convolutional rules. Such rules reduce the computation by avoiding part of the useless operations (null values). The hardware implementation enables the acceleration of dense DNNs without intrusive PE variations.

Eyeriss [132] simply exploits sparsity by clock-gating the PEs with zero value, i.e. not performing the multiplication. Although this reduces the power consumption, highly sparse DNN models could cause a poor PE array utilization. ZeNA [227] was the first zero-aware architecture targeting the CNNs, able to skip ineffective computation induced by both weights and activations. Moreover, it addresses the unbalanced workload among PEs due to the zero-skip operation by introducing a novel load distribution method.

TABLE 5. Summary table of sparse architectures.

Architecture	Target device	Compression Format	Compressed Data	Zero Skip
Eyeriss [132]	ASIC	RLC	A	A
Cnvlutin [222]	ASIC	CSR	A	A
Cambricon-X [113]	ASIC	CIS-alike	W	W
SCNN [223]	ASIC	CSC	A+W	A+W
EIE [224]	ASIC	CSC	W	A+W
NullHop [225]	FPGA	CIS	W	A+W
SqueezeFlow [226]	ASIC	none	none	W
ZeNA [227]	ASIC	none	none	A+W
Huan et al. [228]	ASIC	none	none	A+W

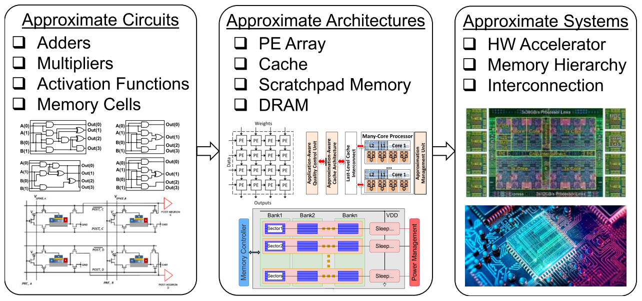


FIGURE 50. Opportunities for employing approximate computing in deep learning.

Huan *et al.* [228], instead, proposed an approximate architecture that skips near-zero multiplications, providing a further reduction of computation (1.92x over LeNet5) with negligible accuracy loss.

Table 5 summarizes what has been discussed in previous paragraphs about the sparse architectures analyzed in this section. It reports the data compression format and which data among activations (A) or weighs (W) is subject to the compression. Besides, the last column reports the type of data for which unnecessary operations are skipped.

I. APPROXIMATE COMPUTING FOR DEEP LEARNING AND THEIR RESILIENCE

Approximate computing is a well known paradigm, whose basic idea is to trade quality for efficiency, at different abstraction levels [229]. Therefore, it is desirable for non-safety-critical tasks, or for applications that are resilient to approximation errors [230]. Many studies have analyzed its applicability on DL-based applications [231]. An overview of the possibilities of employing approximate computing is shown in Figure 50.

The most compute-intensive operations that are performed in the inference are the multiplications. Approximate multipliers can be employed in DNN accelerators to reduce the power consumption [232].

At the architecture-level, systematic resilience analyses are needed for applying approximate computing in CNNs [233] and CapsNets [234]. The error generated by approximate MAC units in systolic array-based DNN accelerators can be mitigated by employing curable approximations [235]. This is extremely useful for reducing the critical path and energy consumption of the DNN accelerators, without sacrificing the classification accuracy. AxTrain [236] is a framework for DNN training that enables approximate inference. Otherwise, a layer-wise approximation of DNN accelerators at the inference stage can be done automatically [237]. CAxCNN [238] is a methodology for approximating the filter weights of DNNs without retraining and executing DNN inference with low-complexity multipliers.

Approximate memories [239], [240] can further reduce the energy consumption of DNN accelerators and systems. The work in [241] optimized the communication network for reducing the computational cost of DL training and inference.

A cross-layer approach [242] leads to integrate approximate computing in a compression framework for further reducing the energy consumption of DNN accelerators.

J. EMBEDDED VS CLOUD COMPUTING

So far, the focus has been mainly on the development of embedded architectures for deep learning. However, it is also necessary to mention the other solution that is gaining ground, namely *cloud computing*. Cloud computing is a paradigm of service delivery, especially storage of data and computational resources, offered by a provider to a client through the Internet.

Cloud computing offers some advantages when applied to deep learning. As demonstrated in the previous paragraphs, deep learning is based on the availability of a large amount of data, especially during the training phase. For the latest models of neural networks, many computational resources are also required. These resources may not be available and accessible to everyone, so services provided by third parties may be a valid solution. Besides, cloud services have a very flexible availability of resources, which can be scaled during the development of a project to better adapt to different needs. Finally, many cloud computing services for AI and machine learning offer solutions and resources that do not require in-depth technical knowledge, allowing even inexperienced people to approach this field and exploit its potential. There are currently several providers offering cloud computing solutions, among them Alibaba Cloud [243], Amazon Web Services (AWS) [244], IBM Cloud [245], Google Cloud [246], Google Colab [247] and Microsoft Azure [248].

However, cloud computing has some disadvantages that do not make it suitable for all applications. First of all, cloud computing is based on the availability of an Internet connection, and it is therefore not ideal for applications that do not permit interruptions of service, such as self-driving vehicles. Data transmission between client and server is also subject to security issues as it is more vulnerable to breaches. Cloud computing is, therefore, not suitable for applications where there are strict regulations on data security, such as government or defence services. Finally, in latency-constrained applications, such as again self-driven vehicles or virtual reality applications, it is preferable to have near-sensor computing rather than relying on the transmission of data via the

Internet. However, the advent of 5G could potentially mitigate this problem.

K. SNNs HARDWARE ACCELERATORS

Modern computing systems based on the Von Neumann architecture are not efficient for the SNNs implementation, because of the physically separated computational and memory units [249]. Therefore, novel computational architectures are necessary to implement SNNs with high performance and low energy. Several accelerators for SNNs have been proposed in the literature. The most popular ones are adopting the *neuromorphic* architecture.

SpiNNaker [61] is a system designed to implement large SNNs in real-time. Using ARM9 cores as building blocks, it implements event-driven computation and communication, interfacing with Python libraries such as PyNN. Its second version, SpiNNaker 2 [250], increases the number of cores for implementing deep learning with sparse connectivity.

IBM TrueNorth [60] is designed with a 28-nm CMOS technology, with 4,096 neurosynaptic cores. Each core has 12.75 KB of local SRAM and can support up to 256 neurons. The scaling and integration of multiple chips is allowed by the spike-based nature of the communication and routing infrastructure in an asynchronous-based NoC.

Intel Loihi [63] provides highly parallel and power efficient asynchronous computations. The chip implements in a 14-nm CMOS technology a mesh of 128 neurocores, each of them having 1,024 spiking neurons and 2 Mb of SRAM. Scaling is possible through a hierarchical connectivity between chips. Moreover, several neuromorphic learning rules are supported.

BrainScaleS [62] is a mixed analog-digital system, with analog neurons and digital communication. Like SpiNNaker, it also allows PyNN interface.

IV. MEMORY HIERARCHY

While optimizing the algorithms and accelerating the implementation of computational primitives is of fundamental importance to achieve the best performance, inefficient memory management could undermine all efforts made to achieve high throughput and energy efficiency claimed by the accelerator design [251]. Typically, memory accesses dominate the energy consumption of a system [252]. As depicted in Deep Compression [203], [253], using a 45 nm technology, a 32-bit adder consumes 0.9 pJ, while SRAM and DRAM access require respectively 5.5× and 711× more energy. In such a situation where the storage elements constitute a clear efficiency bottleneck, memory must be taken into account from the earliest design steps as a first-order concern.

Conversely to processors, where the general-purpose structure prevents adaptation to the workload, for other platforms it is possible to make a tightly tailored design on the specific algorithm in order to reduce at minimum the memory transfer. These considerations are crucial, especially in the field of machine learning, where the enormous number of MACs to be performed requires an enormous and continuous data movement towards the processing units. Considering, for example, a 1G fully connected layer running at a typical video recording frame rate (30 fps), using the above DRAM technology, its computation would require (30 fps)(1G)(640 pJ) = 19.2 W that is a considerable amount of power, unaffordable for mobile devices.

1) INFERENCE vs. TRAINING

From a memory perspective, training is way more intensive than inference. While in the latter the NN is crossed only once, in the former the backpropagation mechanism imposes to cross it backward, reloading both activations and weights. Thus, the training has an almost double cost. Generally speaking, in most of the industrial, medical and commonly used applications, there is no reason for on-line training. Usually, neural networks are trained on a dataset off-line and then delivered to the end-users. As the dataset is periodically improved by adding corner cases, networks can be realigned through a new off-line training session. Since such an operation is performed off-line, there is no need for highly optimized hardware platforms, but rather for high-speed general-purpose architectures, such as GPUs, able to scale down the training time with no constrains over the power envelope. Moreover, even though ML researchers are very interested in speeding up learning, from a business perspective, this represents a small market. According to the above, and knowing that the nature of the computations required to carry out the backpropagation and the inference is almost identical, from now on we mainly focus on the inference stage that offers more case studies.

As mentioned before, accelerating large size ML algorithms involves high memory traffic. Focusing on the currently most known and used networks, DNN and CNN, we analyze in detail their main fundamental layers from a typical processor memory organization perspective, providing an idea of the number of operations to be carried out and the possible optimizations.

2) FULLY CONNECTED LAYER

Among NN layers, FC ones are those which require the highest memory transfer due to their topology. Considering a layer composed of C_i input neurons and C_o output neurons, the synapses (i.e. weights) are represented by a $C_i \times C_o$ matrix. Thus, the execution of the entire layer can be summarized in a matrix-vector multiplication that needs a total number of memory transfers equal to $C_i \times C_o + C_i \times C_o + C_o$ where each addendum represents respectively the inputs loaded, the weights loaded and the output written back to the main storage.

The matrix-vector multiplication is a critical operation, especially in case the weight matrix is larger than the lowest cache level capacity. In such a case, it is impossible to reuse the matrix values, and new memory accesses are performed every cycle. In the case of CPUs and GPUs, this problem can be overcome by batching [254]. This technique allows to group multiple input vectors into a single matrix and reuses the weight parameters. However, real-time applications cannot use this optimization because a certain latency is introduced. Therefore batching can only be used during offline training, where the dataset is provided a priori. As far as inference is concerned, we prefer techniques capable of overcoming the bottleneck represented by memory by spatially and temporally distributing the workload as seen in Section III-A, or compressing the network by reducing the number of parameters as shown in Section III-G. Nevertheless, compression generates irregular patterns that make CPUs and GPUs ineffective.

Theoretically, input activations can be reused for each output one, but unfortunately, their size ranges from few thousands to hundreds of thousands, making them unfeasible to be stored on an L1 cache. Tiling could be used to subdivide the loop over the input neurons. However, it is not possible to perform loop tiling over a factor without affecting the rest of the execution. Indeed, increasing the reuse of the input neurons, the amount of the partial output sums to be stored back to the main storage increases as well. Thus, the input memory bandwidth saved from the tiling is partially compromised by the partial sums write back, but still advantageous. As far as weights are concerned, they are unique for each input, so it is not possible to reuse them. Moreover, in the DNNs these vary from tens of millions to some billions, making it impossible to store them even on higher cache levels.

3) CONVOLUTIONAL LAYER

With respect to FC layers, the convolutional ones are built on a 2D scheme (3D considering the channel direction), exhibiting an input and an output feature map. The input feature map can be reused as many times as the number of kernels. More precisely, since the convolutional windows tend to overlap, the single input feature map windows, with size equal to the kernel one, can be reused $\frac{H_k \times W_k}{S_x \times S_y}$ times as shown in Figure 29, where H_k and W_k are the sizes of the kernel and S_x and S_y are the stride over x and y directions. Consequently, as described above for FC schemes, it is possible to perform a tiling loop over the two dimensions of the IFM with a reuse strategy to reduce the accesses to the main storage. In this case, tiling the input does not affect the output. Consequently, in GPUs and CPUs, no tiling is performed since it is possible to fit an entire kernel volume in an L1 cache; thus, an entire OFM can be produced without the need to break down the loop. Indeed, typical kernels size is $H_k \times W_k \times C_i$, where H_k and W_k are in the order of ten, while the number of input channels (C_i) can reach the hundreds. For FPGA and ASIC approaches, the reuse strategy can be way more aggressive thanks to ad hoc designs as explained in the following. Kernels are usually shared, reducing considerably the number of DNN parameters, and consequently the required bandwidth. Nonetheless, the number of output channels can make the synapses unfeasible to be stored in an L1 cache. Indeed, the weight volume expressed as $H_k \times W_k \times N_i \times C_o$, where C_o is the number of OFMs, can easily exceed the lower level of the memory hierarchy. Also, in this case, it is suggested to use a tiling to break the loop over the output feature map, resizing the total capacity in $H_k \times W_k \times C_i \times TC_o$ sets, where the TC_o is the tile size. In the rare case of not shared weights, as discussed for FC layers, not even the L2 cache could fit them, making reuse impossible.

4) POOLING LAYERS

Unlike the previous layers, pooling has no weights, and the number of OFM is equal to IFM, thus the opportunities to perform data reuse are fewer. The sliding windows, over which the pooling is performed, generally do not overlap, consequently, the bandwidth for input neurons is higher than the convolutional approach. Even with the introduction of the IFM tiling, performance would improve marginally.

Taking into account the three types of layers described above, it is clear that the required bandwidth is profoundly different from each other. Figure 51, for example, shows the bandwidth needed for the execution of AlexNet on a device able to perform 100 Gops/s with 100% efficiency, i.e. no stall and data dependencies, with no memory constraints. Despite the bitwidth for both activations and weights is just 16 bits, the bandwidth for some layers, especially FC and max pooling, where data reuse is practically impossible, is unattainable from any commercially available memory. This once again highlights how difficult it is to execute ML algorithms efficiently, in particular on devices with rather modest hardware, as may be the case with IoT nodes, which are mainly CPU-based. The architecture of these nodes must guarantee flexibility and speed of execution for a wide range of algorithms, therefore it cannot be optimized for the DL. Researchers over the years have tried to improve the libraries and kernels of basic operations carried out on their processors to maximize the management of storage elements such as the Intel MKL-DNN [255] and ARM CMSIS-NN [256]. The former library works on the data format mapping multidimensional arrays into linear memory address spaces. Moreover, it enables lower numerical precision primitives, accelerating the execution of multiple operations, i.e., increasing the number of operations per second, and enhancing the performance of the cache at bandwidth parity. The latter intends to reduce memory overhead and maximize NN execution on Cortex-M processors for low-power applications oriented to IoT devices. Another example is represented by Garofalo *et al.* [257], who proposed PULP-NN, a library designed for a parallel cluster of tightly-coupled RISC-V processors. Its set of software kernels targets the inference of quantized NN, being capable of exploiting sub-byte bitwidth data.

What just said for CPUs is also valid for GPUs, with the big difference that their capability to parallelize large workloads makes these devices ideal for DNN applications, although expensive from the power point of view. NVIDIA developed the CUDA Deep Neural Network library (cuDNN) [100], a special library that also includes the possibility to use a fixed point format at 16 and 32 bits, moreover, transforms convolution operations into multiplications between matrices, which are extensively optimized. This property is reflected in

AlexNet						
Layer		Input Feature Map size	Kernel size	Output Feature Map size	Stride	Activation
1	conv	227x227x3	11x11x3x96	55x55x96	4	relu
	max pool	55x55x96	3x3	27x27x96	2	
2	conv	27x27x96	5x5x96x256	27x27x256	1	relu
	max pool	27x27x256	3x3	13x13x256	2	
3	conv	13x13x256	3x3x256x384	13x13x384	1	relu
4	conv	13x13x384	3x3x384x384	13x13x384	1	relu
5	conv	13x13x384	3x3x384x256	13x13x256	1	relu
	max pool	13x13x256	3x3	6x6x256	2	
6	FC	9216		4096		relu
7	FC	4096		4096		relu
8	FC	4096		1000		relu

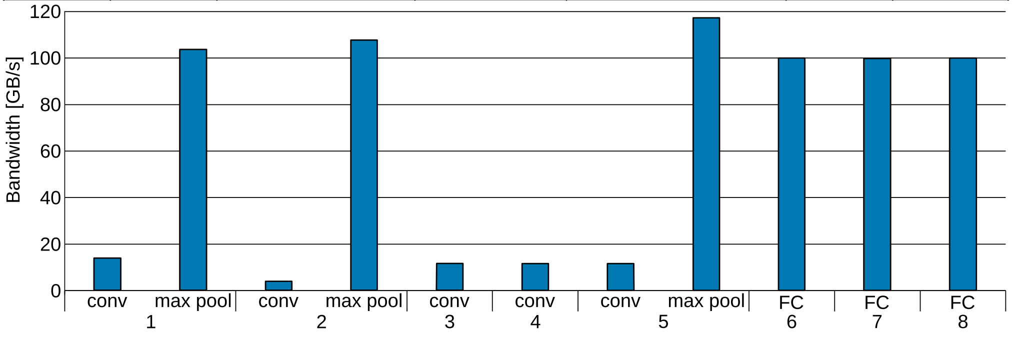


FIGURE 51. Example of required bandwidth per layer executing Alexnet on a device able to perform 100 Gops/s with 100% efficiency.

a reduction in the demand for RAM and a consequent increase in the number of supported operations. However, for large DNN models, it is essential to tune the memory usage to fit them into the DRAM. vDNN [258] virtualizes the memory of the CPU and GPU so that it can be simultaneously used for training in a hybrid fashion. Kim *et al.* [259] extended the concept of vDNN to a multi-GPU environment employing PCIe-bus. Furthermore, thanks to a prefetching algorithm, they can increase the mini-batch size of 60%.

Conversely to GPUs, FPGA and ASIC accelerators have a limited amount of memory. In CNP [260] for example, in order to accommodate a large number of DSPs on a Virtex4 SX35 FPGA platform, the authors designed an interface with an external memory capable of performing 8 read/write operations. However, their flexibility and the possibility to design their memory hierarchy tailored to the specific problem can lead to a lower energy envelope. The sizing of on-chip memory buffers is not trivial and depends on many factors such as layer size, layer type, frequency of buffer usage. Wei *et al.* [261] have proposed an FPGA-based layer conscious framework to allocate on-chip buffers efficiently. Such a paradigm combined with buffer sharing saves resources and enhances their usage. Since DRAM has an access cost about 130× higher than SRAM, in some cases, it has been thought to directly remove this storage device as in the case of Park and Sung [262]. Exploiting fixed-point data format and the capability of NN to work even in case of reduced precision [203], Park *et al.* were able to fit the entire DNN model into the on-chip memory, reducing the power consumption drastically. Following the same approach Du *et al.* have proposed ShiDianNao [129], an ASIC designed to be integrated into a commercial image chip typical of smartphones. Being in close contact with the sensor, the data it processes is taken directly from the local SRAM, minimizing the power needed. ShiDianNao is the last accelerator of the series started with DianNao [24], a small-footprint memory-wall aware accelerator for large NN models, and continued with DaDianNao [134]. The latter instead proposes a multi-chip ML architecture with 64 cores in supercomputer style able to achieve a speedup of about 450x over a typical GPU. While the above architectures distribute the on-chip memory among the PEs, i.e., near computation, there have also been efforts to do the opposite, namely to bring the computation into the storage elements. This is the case of the logic-in-memory (LIM), where easy computational tasks are executed directly inside the memory like in [263]–[266].

V. DEEP LEARNING SECURITY

Despite the great success and popularity of deep learning in recent years, recent researches showed that DNNs have intrinsic weaknesses that can threaten the security [267], [268] [269]. Starting from the work of Goodfellow *et al.* [270], many researches have been conducted, with the purpose of identifying weaknesses (*Adversarial Attacks*) and their countermeasures (*Adversarial*

Defenses) [271], [272]. Moreover, machine learning models can be stolen [273] or inverted [274].

A. ADVERSARIAL ATTACKS

The basic idea behind an adversarial attack is to make a machine learning model classify a malicious sample wrongly. In case of image classification, the adversarial attack introduces a noise in the input image to create the adversarial example, which is classified wrongly by the DNN. Adversarial attacks can be categorized according to different attributes, e.g., the choice of the class, the kind of the perturbation and the knowledge of the network under attack [275], [276]. We summarize these properties in Figure 52.

The goal of an adversarial attack is to be at the same time imperceptible and robust [277]. A successful adversarial example should not have obvious variations perceived from an human eye, compared to the original image. Moreover, an attack is robust if the gap between the probabilities of the adversarial class and the correct class is so large that, after a transformation (e.g., noise filtering, compression or resizing), the misclassification still holds. These kind of attacks have been evaluated also on CapsNets [278] and SNNs [279]. Moreover, if applied on a different domain [280], the imperceptibility of the adversarial examples can be improved.

Several types of adversarial attacks have been proposed.

Poisoning attacks [281], [282] [283] contaminate the training data in such a way that the decision boundaries of the classifier are pushed to incorrect zones, thus reducing its classification accuracy on clean inputs. More specifically, backdoor attacks [284] train a network in a way that, when exposed to a specific noise pattern that plays the role of a trigger, it is fooled. Triggered by an adversarial noise pattern, the NeuroAttack [285] introduces a backdoor Trojan to fool DNNs and SNNs with bit-flips.

Gradient-based attacks like FGSM [270] and its variants [286], [287] [288], [289], [290] are white-box adversarial attacks that perturb the inputs based on the gradient of the output probabilities with respect to the inputs. They only introduce perturbations at the inference stage, without modifying the training data.

The Carlini & Wagner attack [291] aims at minimizing at the same time (i) the distance between the original image and the adversarial image and (ii) the distance between the maximum output activation and the confidence of the target class.

Decision-based attacks [292], [293], [294], [295] are black-box adversarial attacks which estimate the decision boundary and aim at crossing it to obtain a misclassification. The quality of such attacks is measured in terms of number of queries, i.e., the inference passes with different inputs.

Universal perturbations [296] aim at identifying a noise pattern, specific for a given dataset, which, when added to the input, significantly reduces the test accuracy of any deep learning model.

B. ADVERSARIAL DEFENSES

Several defense methods have been studied and proposed. They aim at increasing the generalization of DNNs, while they perform better against different types of attacks. However, one of the main drawbacks of applying the defenses on DNNs is that the classification accuracy on clean images decreases.

Data protection defenses [297], [298] analyze the impact of the input in order to identify the noise, thus effectively working against poisoning attacks.

Standard DNN compression techniques has been adapted to successfully defend against adversarial attacks. Fine-Pruning [299] removes the redundant connections in DNNs which do not significantly contribute for the accuracy of the clean data in order to remove the effect of the backdoor. A quantization-based defense [300] reduces the success rate of the attack by quantizing the input pixel intensities.

Adversarial training [289] is the de-facto standard defense method against adversarial attacks. Since the adversarial examples are added to the training set, the classifier is able to learn these perturbations. As a drawback, the adversarial training adds a prohibitive overhead in the training process. Further variants of such defense [301], [302], [303] aim at reducing its computational cost and training time overhead.

Different pre-processing techniques can elude the effectiveness of adversarial attacks. Simple pre-processing filters [304] completely alter the functionality of the attack. Randomized smoothing [305] produces a Gaussian noise at the input to mitigate the effect of the adversarial perturbations on the inputs. It has been demonstrated to be effective also on large perturbations on large and complex datasets.

Detectors [306] add a sub-network model to detect whether an input is an adversarial example or not. This algorithm can be successfully executed in specialized hardware and integrated with DNN accelerators [307].

VI. BENCHMARKING

Since Deep Learning is a critical topic in the research community, over the years, big companies have made available a massive series of tools to help the development of new models. These frameworks, in addition to the most recent and updated datasets, are crucial for both software and accelerator development. The possibility to explore new models and evaluate them in terms of workload, the trade-off between complexity and accuracy, access to memory, numerical representation (floating-point vs fixed-point) is a fundamental step in the hardware design phase. All these steps are of great importance to understand what the performance of the accelerator will be. In this section, we present the main frameworks for the DL and the datasets used to determine the performance of the algorithms. Finally, parameters and metrics for the comparison of hardware platforms are discussed.

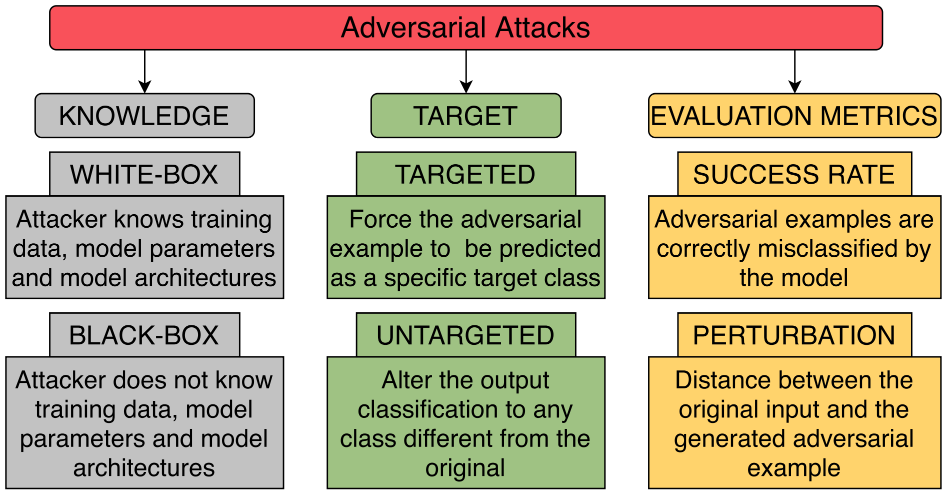

Adversarial Attacks
KNOWLEDGE
WHITE-BOX
Attacker knows training data, model parameters and model architectures
BLACK-BOX
Attacker does not know training data, model parameters and model architectures
TARGET
TARGETED
Force the adversarial example to be predicted as a specific target class
UNTARGETED
Alter the output classification to any class different from the original
EVALUATION METRICS
SUCCESS RATE
Adversarial examples are correctly misclassified by the model
PERTURBATION
Distance between the original input and the generated adversarial example

FIGURE 52. Taxonomy of Adversarial Attacks.

A. FRAMEWORKS

Frameworks are working environments that provide the developers with all the basics and support to build new ready-to-use models, as depicted in Table 6. Having such tools, able to compose a DNN using high-level programming language like Python and then test the performance of the algorithm, speeds up the research work enormously. Moreover, profiling the code execution and understanding where the critical load is located, it is possible to define which parts will have to be translated into hardware and, consequently, what needs to be accelerated.

In the case of CPUs and GPUs instead, libraries and frameworks are essential to parallelize and distribute the effort among the cores.

To be noted that many frameworks transform DNN models into optimal graphs. This is only an effective method for visualizing the operations to be performed sequentially.

1) TENSORFLOW [98]

Google's Tensorflow is one of the most popular DL framework. It supports many different languages such as JavaScript and Java, C++, Go, C#, Julia, even though the most convenient client remains Python. It is characterized by a static computation graph, which means that it first defines the graph, and then processes it. Since the model is static, it is not possible to make changes in the structure at run time, but it is necessary to do the training of a new structure. The efficiency of its primitives compensates this lack of flexibility. It is optimized for Tensor Processing Unit (TPU) architectures.

2) PyTorch [97]

Created by Facebook, it is the principal competitor of Tensorflow. Unlike the previous one, PyTorch takes advantage of a dynamically updated graph. This means that it is possible to make changes to the DNN architecture on the fly. Generally speaking, PyTorch is often used in projects in which new training paradigms are exploited. In fact, the dynamic graph property is exploited during the backpropagation task, where the normal graph execution needs to be altered. Moreover, it supports different data parallelism (suitable for hardware solution exploration) and distributed learning models.

3) CAFFE [99]

This DL-based framework supports C, C++, Python, and MATLAB. It is mainly used to model CNNs. In its repository called Caffe Model Zoo, it is possible to access a wide range of pre-trained models ready-to-use. Thus, whenever there is a problem with image processing, Caffe could be the solution. Since its libraries are mainly written in C++, its strength lies in the speed of execution. However, unlike other frameworks, Caffe does not allow a fine-granularity network layer alteration by the user, which makes it inflexible. Moreover, for recurrent model applications such as the Natural Lenguage Processing, the available resources are poor.

4) MXNet [308]

This work environment supports a wide range of languages including C++, Python, R, Go, JavaScript and Julia. The strength of this framework lies in its ability to parallelize execution both on multiple GPUs and on multiple machines, as in the case of Amazon servers.

5) CHAINER [309]

It is the first framework to exploit the dynamic computation graph, allowing for varying length input, a handy feature in problems of natural language processing. Chainer is built on Numpy and Cupy libraries and is completely written in Python. Since it is faster than other Python-based frameworks, today it is the leading tool for GPU performance in data centres.

6) MICROSOFT COGNITIVE TOOLKIT [310]

This framework, also known as CNTK, supports Python, C++ and command-line interface. Unlike Caffe, when a new layer model is needed, it can be built thanks to the fine granularity of the base blocks, without the need for low-level code. Concerning the operation over multiple machines, it presents higher performance compared to Theano and Tensorflow. However, as a result of a lack of support related to ARM architectures, the applications over mobile devices are limited.

7) PaddlePaddle [311]

This is an industrial-oriented framework equipped with basic libraries and tools for end-to-end product development. It mainly supports CNNs and recurrent neural networks for highly optimized computation and memory recycling. Moreover, it can efficiently scale over heterogeneous architectures to speed up the training process.

8) ONNX [312]

This is not a framework but a representation format for deep learning models. Microsoft and Facebook collaborated to create such a format in order to make the models portable from a framework to another. In some cases, it is convenient to perform the training on a platform and the inference on another. Moreover, ONNX can also be a valuable resource for developers, researchers and the open-source world, in fact, any pre-trained model can be shared with the community, and every user can choose the most suitable framework. It is supported by TensorFlow, PyTorch, Caffe2, Chainer, MXNet, Keras, Microsoft Cognitive Toolkit, PaddlePaddle, and many others.

9) KERAS [313]

Keras is an Application Programming Interface (API) for ML and DL. It can be used in many of the shells presented above. It is a high-level code abstraction for implementing NNs exploiting the lower level primitives of the corresponding framework. Working at a higher level, it is suitable for fast prototyping and handling wide amounts of data streams thanks to Python generators and serialization/deserialization APIs.

B. DATASETS

As the frameworks are used to build new DL models, datasets are fundamental to test their performance concerning the designed task. It is essential to underline that there are countless datasets for each specific task (image classification, object detection, etc.). However, datasets of the same task are hardly comparable to each other, the difficulty of each could vary in orders of magnitude as depicted in Table 7. Considering, for example, MNIST and CIFAR100, both datasets for the image classification, the first is a collection of handwritten digits in grayscale, while the second ranks objects in 100 different classes. Typically, different datasets reflect different DL models. The more complicated the dataset, the greater the model size in terms of weights and consequently in the number of operations (MAC). The metrics used to evaluate the performance of DL models on datasets are mainly two: accuracy in Top-1 and Top-5 mode, and weights size. Top-5 means that if in the 5 classes with the highest score there is the correct one, then it is counted as correct. Top-1 instead needs the highest score class to be the correct one.

1) MNIST [29]

This dataset is composed of 70,000 images divided into 10 classes representing handwritten digits. 60,000 are for training, while the remaining are the test set. Each image is 28×28 pixel size in grayscale.

2) ImageNet [39]

This dataset is composed of 1.3M training images, 100,000 for test and a final 50,000 for validation. All the images are divided into 1000 classes organized according to the WordNet. This latter allows managing synonyms and ambiguities. Each image has a size of 256×256 pixels in color. ImageNet is the core of a famous challenge where DNNs and CNNs try to score the best Top-1 and Top-5 accuracy.

3) CIFAR [314]

Under the name of CIFAR two different datasets fall, namely CIFAR-10 and CIFAR-100. Both are composed of 60,000 32×32 pixels coloured images, but while the former ranks them in 10 classes, the latter uses a finer classification in 100 classes. Both datasets have 50,000 images for training and the remaining for test purposes.

4) COCO [315]

This dataset aims to advance the object recognition state-of-the-art by putting together also segmentation and captioning. It is composed of 328,000 images of everyday scenes for a total of 91 stuff categories and 80 object classes.

5) OPEN IMAGES V6 [316]

This dataset contains 9M images equipped with labels, objects bounding boxes, segmentation, narratives and relationship views. Each image contains 8.3 objects on average. With 16M bounding boxes over 600 categories, it is the largest object localization dataset ever realized. Moreover, it is able to provide 19,957 classes with an annotation at the image-level.

6) CORe50 [317]

This is the first collection of images designed for continual object recognition, i.e., learning new classes online. It is composed of 11 sessions of 300 RGB-D images that can be classified by objects (50) or by categories (10). The objects are held and moved by the operator who is recording a 15 seconds video at 20 fps (300 frames in total) from a subjective point of view.

7) ObjectNet [318]

ObjectNet is a dataset composed only of a test set of 50.000 images. The aim is to test object recognition

TABLE 6. Framework summary.

Name	Year of creation	Language	Creator	Features
Tensorflow [98]	2015	JavaScript, Java, C++, Go, C#, Julia, Python	Google	- Static computation graph - optimized for TPU
Pytorch [97]	2016	Python, C++, Java, CUDA	Facebook	- dynamic computation graph - different data parallelism
Caffe [99]	2016	C, C++, Python, MATLAB	Berkeley	- speed of execution - fine granularity layer not allowed
MXNet [308]	2017	C++, Python, R, Go, JavaScript, Julia	Apache Software Foundation	- parallelized execution on multiple GPUs and machines
Chainer [309]	2015	Python	Seiya Tokui	-dynamic computation graph -varying length input support
Microsoft Cognitive Toolkit [310]	2016	Python, C++	Microsoft	- fine granularity layer basic block management
PaddlePaddle [311]	2016	Python, R, Go	Baidu	- industrial-oriented - heterogeneous architecture-scale training process
ONNX [312]	2017	C++, Python	Facebook, Microsoft	- representation format for DNN - portable
Keras [313]	2017	Python	Google	- Application Programming Interface (API)

applications in a real-world scenario in which images present background, rotation and often the viewpoint are random. ObjectNet showed that applications performing at the top in their respective datasets present a lack of generalization with a 40-45% drop in the performance. Fine tuning-robust, this dataset represents one of the best challenges for the generalization of object recognition tasks.

C. NEURAL NETWORKS MODEL METRICS

The attributes of DL models can be evaluated, considering a few important metrics.

- **Accuracy**. The accuracy (Top-5 or Top-1) of a model with respect to a specific Dataset is an important metric. Besides the dataset, the training properties must be reported, e.g., the number of epochs, learning rate, data augmentation.
- **Model Architecture**. The shape of the DL model is the foundation for understanding how it operates. The number and type of layers, the size of feature maps and the number of channels, the number of filters and their size are all fundamental properties to understand how an algorithm elaborates the incoming raw data.
- **Workload**. The size of the input feature map and the number of kernels define the total amount of operations (MACs) to be performed. The MAC count is one of the basic metrics to evaluate a DNN, and it also defines the throughput and the energy effort of the target hardware platform. To be noted that as explained in Section III-H, only effective MACs should be counted.
- **Memory requirement**. The amount of weights (non-null) determines the storage impact of the model. A large number of weights could represent a limit for the target hardware platform and for the power envelope as well.
- **Training time**. Typically, the higher the complexity of the model, the more accurate it is. On the other hand, complexity results in difficulties in training the model. In fact, the more the weights and the number of layers, the more epochs will be needed. This metric can be expressed as number of training epochs or GPU hours related to a specific dataset to obtain a certain accuracy.
- **Adversarial Robustness**. As explained in Section V, DNNs are vulnerable to adversarial attacks, which is one of the hottest research topics in the development of new deep learning models. Therefore, it is of fundamental importance to provide the model with defense algorithms and perform an exhaustive and correct evaluation of adversarial attacks.

D. HARDWARE ACCELERATOR METRICS

The fundamental metrics to evaluate the hardware platforms are:

- **Power**. The power consumption of the device determines the final application for which it can be exploited. In addition, an important metric is energy efficiency defined as pJ for MAC. Note that the power consumed must also include that spent on readings from the off-chip memory, as explained in Section IV.
- **Throughput**. Throughput and latency depend directly on the working frequency of the device and the memory bandwidth. Such metrics are crucial to define how often the hardware platform can perform a complete inference or backpropagation of a model. The throughput is often defined as billions of operations per second (Gop/s) or as billions of MAC per second (GMAC/s) where 1GMAC/s $\simeq$ 2Gop/s.
- **Area**. From the area of the device, cost and integration capacity in larger systems are derived. On the other hand, the area depends on the technological node and the amount of memory. Memory, as in the case of power, plays a critical role also for the area. This represents another reason to optimize its use.

TABLE 7. Dataset summary.

Dataset	Format	Instances	Size	Task	Year of creation	Creator
MNIST	Images, text	60,000	50MB	Classification	1998	LeCun et al. [29]
ImageNet	Images, text	14,197,122	150GB	Classification, Object recognition	2009	J. Deng et al. [39]
CIFAR	Images, text	60,000	170MB	Classification	2009	A. Krizhevsky et al. [314]
COCO	Images, text	2,500,000	25GB	Object recognition	2015	T. Lin et al. [315]
Open Image V6	Images, Segmentation masks, text	9,000,000	500GB	Classification, Object recognition, Object detection, Object segmentation	2020	Kuznetsova et al. [316]
CORe50	Images	3,300	25.8GB	Continuous Object recognition	2017	Lomonaco et al. [317]
ObjectNet	Images, text	50,000	-	Object recognition	2019	Barbu et al. [318]

In addition to the main metrics listed above, there are others that could be defined as application-dependent. For example, the flexibility with which an accelerator can be adapted for more complex models, parallelizing its architecture, or how a device can scale with respect to the required computing accuracy, having a tuning bitwidth. Although metrics are easily definable, comparisons between different hardware platforms are not always straightforward. There are many factors on which possible comparisons depend, moreover the benchmarks used to evaluate performance are not always impartial. It is, therefore, necessary to assess all the side-factors on a case-by-case basis.

VII. CHALLENGES AND THE ROAD AHEAD

As discussed in Section I, AI and DL are adopted in numerous and various fields, and the number of their applications is growing over time. The number of investments that have been made in AI startups over the years is a clear demonstration of this growth. In 2010, investments amounted to \$1.3B, while in 2018, they reached \$40.4B [319]. The average annual growth rate was 48%, and this trend does not seem to stop. Many technical reports [319] [320] indicate that, in the years to come, AI will be a driving force to the economy.

The development and diffusion of AI applications are closely related to technological advancements, i.e., the chips. There is a flow that runs from the application, that is expressed as an algorithm. The algorithm is deployed on a chip, which consists of some devices realized in a technology, e.g., CMOS technology (see Figure 53). The growth of AI applications in number and complexity has required more and more performance from the hardware (*application-driven development*). Before 2012, the chip compute doubled about every two years. After 2012, the year of DL boom, AI chips started to double the compute every 3.4 months [319]. On the other hand, the development of new technologies and hardware improvements allowed to develop more complex and therefore accurate applications (*technology-driven development*). The two directions of development continue to feed each other in a virtuous circle. It is estimated that, in 2025, the AI chip market will reach a value of \$29B, while in 2017, it was only \$2B [320].

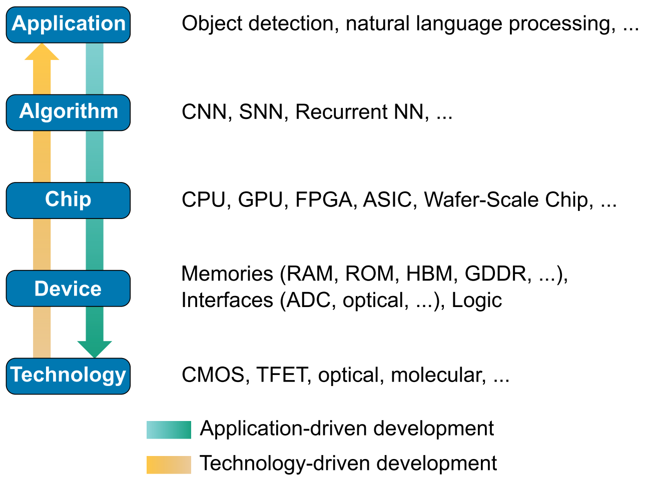

Application
Object detection, natural language processing, ...
Algorithm
CNN, SNN, Recurrent NN, ...
Chip
CPU, GPU, FPGA, ASIC, Wafer-Scale Chip, ...
Device
Memories (RAM, ROM, HBM, GDDR, ...), Interfaces (ADC, optical, ...), Logic
Technology
CMOS, TFET, optical, molecular, ...
Application-driven development
Technology-driven development

FIGURE 53. Development flows that run from the application to the technology and vice-versa.

To maintain such a high growth rate, the industrial and academic world will face new challenges in the coming years, which we summarize in the following, together with the possible solutions, research trends and future directions.

1) VON NEUMAN BOTTLENECK

One of the biggest challenges the developers are facing currently is the Von Neuman bottleneck, i.e., the bandwidth that modern memories can provide is not sufficient for the huge amount of data that AI chips need to process. To get around the problem, it is possible to modify the algorithms to reduce the number of data items to be used (e.g., model compression, pruning, or quantization).

To solve the problem, it is necessary to act at the memories level. One possible solution is the increase of the memory bandwidth, and this is the purpose of High Bandwidth

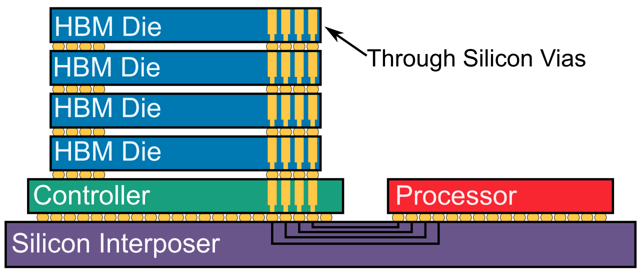


FIGURE 54. High Bandwidth Memory (HBM) scheme.

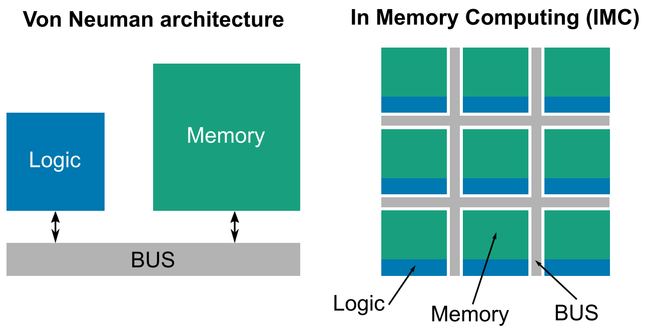


FIGURE 55. Comparison between the structures of a Von Neuman Architecture (left) and an In Memory Computing (IMC) architecture (right).

Memories (HBMs) (see Figure 54). HBM is a stacked DRAM integrated with the processing elements through a silicon interposer. A single HBM2 block has a bandwidth of 256 GB/s, lower than the 616 GB/s bandwidth of a more traditional Graphics Double Data Rate 6 (GDDR6) memory. However, a stack with four HBM blocks reaches a 1 TB/s bandwidth. HBM2 memories are currently used in the Nvidia V100 and P100 GPUs.

Another possibility is in-memory computing (IMC), which consists of moving the logic inside the memory. IMC is particularly suitable for the DNNs operations since DNNs algorithms are deterministic, and it is possible to know when and where data items will be required in advance. IMC wants to enhance DNN acceleration by reducing the latency and power needed to access the memory hierarchy in traditional Von Neuman architectures. Moreover, it increases the parallelization by working with all the memory cells simultaneously. Researchers are currently studying the application of IMC to DNNs algorithms and obtaining promising results [190], [263], [321], [322], and Mythic startup produces IMC accelerators for AI with a 40 nm process.

2) CMOS TECHNOLOGY LIMITATIONS

From the '60s onwards, CMOS technology has been scaling following Moore's Law, according to which the number of transistors on a chip doubles every 24 months. However, this pace of scaling is beginning to stop and it will not be sustainable in the future for technological and economic reasons [323]. Researchers are currently exploring new physical possibilities and a lot of effort is placed in the emerging memories, such as Phase Change Memories (PCMs) [324], [325], Spin-Torque-Transfer Magnetoresistive RAM (STT-MRAM) [326], [327], or Resistive RAM (ReRAM) [328], [327]. Beyond emerging memories, several new technologies are being studied, such as Tunnel FETs, organic FETs, molecular transistors, and spintronic devices. Despite the possible gains deriving from moving to a beyond-CMOS technology, replacing CMOS technology with emerging ones will not be an immediate procedure since it is considered very reliable and easy to manufacture. Moreover, foundries and production lines have been calibrated to this technology and cannot be dismantled until production has paid for the initial investment.

3) AI TOOLCHAINS

Besides the special-purpose ASICs and the programmable CPUs/GPUs, flexible hyper-scale AI accelerators are gaining importance, e.g., Google TPUs or Cerebras Wafer Scale Engine. As seen in Section VI-A, there are several high-level frameworks, mainly python-based, for the description of DLs algorithms. However, there is not yet a unified method to program AI accelerators from a unified high-level language. So far there are the compiler toolchains for CPUs and GPUs, and there is the synthesis toolchain for the FPGAs. The development of a toolchain for AI accelerators programming will be a huge step forward for their diffusion.

4) GENERAL AI

Even though DL models can perform various tasks at a better-than-human level, e.g., object detection or language processing, AI is to be considered still at an early development stage. Indeed, scientists are very far from the so-called Artificial General Intelligence (AGI), e.g., an algorithm able to perform multiple tasks and of taking decisions. Even if an AGI algorithm existed, at the moment, probably, hardware systems could not provide enough computational power for its deployment. For a long future, it will be necessary to combine different algorithms to perform complex tasks. For this reason, it will be important to develop hardware platforms able to support multiple algorithms, easily programmable or reconfigurable.

5) AI AT THE EDGE

It has been listed in the 2020 Top Technological Trends [329] by the IEEE Computing Society. Thanks to the diffusion of 5G connectivity and IoT sensors, ML algorithms will spread into the edge devices. If compared to AI cloud platforms, the edge devices have completely different requirements. During the development of AI edge devices, the focus must be placed on low power and low latency. For this purpose, many roads can be taken. At the application level, it is necessary to develop models co-optimized with the hardware for a more efficient resource handling. At the hardware level, new possibilities are being explored beyond the traditional low-power techniques, e.g., moving the computation in the analog part of the circuit to save energy [330]–[332].

Presently, most of the AI edge devices perform the inference only. The collected data must be sent to the cloud for model training (see Figure 56 top). In the future, it will be

TABLE 8. Comparison among state-of-the-art surveys.

Reference	Year	Citation #	DNN and general ML							SNN		Security	Benchmarking			CPU/GPU/FPGA/ASIC Comparison
			Techniques	Dataflow	Tools for Design	Quantization	Sparsity	Approximate Computing	Memory Hierarchy	Techniques	Architectures	Adversarial Attack	Frameworks	Datasets	Metrics	
[336]	2016	2	≈	×	×	×	×	×	×	≈	≈	×	×	≈	×	×
[333]	2017	-	✓	≈	×	×	×	×	≈	✓	✓	×	✓	×	×	✓
[92]	2017	247	✓	✓	×	✓	≈	✓	✓	×	×	×	✓	✓	✓	✓
[337]	2017	8	✓	×	×	×	×	×	×	×	×	×	×	×	≈	×
[338]	2018	2	×	×	≈	×	×	×	×	×	×	×	×	×	≈	✓
[339]	2018	3	×	×	×	×	×	×	×	×	×	×	×	✓	✓	✓
[340]	2019	14	✓	×	×	×	×	×	✓	×	×	×	×	×	×	×
[341]	2019	11	✓	×	×	×	×	×	×	×	×	×	×	×	≈	×
[342]	2019	7	✓	×	×	×	≈	×	×	×	×	×	×	×	×	×
[343]	2019	-	✓	×	≈	×	×	×	×	×	×	×	×	×	≈	≈
[344]	2019	82	✓	✓	✓	×	×	×	×	×	×	×	≈	≈	×	×
[334]	2020	1	✓	✓	×	≈	✓	×	✓	×	×	×	×	×	≈	×
[335]	2020	-	✓	✓	×	✓	✓	≈	✓	×	×	≈	✓	×	×	≈
[345]	2020	-	≈	×	×	×	≈	×	×	≈	×	×	×	×	≈	×
[346]	2020	-	✓	×	×	×	≈	×	×	×	×	≈	✓	✓	×	×
our work	**2020**	**-**	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

important to move the learning to the edge device for several reasons. The learning in the sensors guarantees real-time lifelong learning, i.e., the device can immediately learn from every sample received and adapt the model consequently. The connection to the cloud is no more needed continuously and higher data privacy can be guaranteed since it is no more necessary to communicate the data but only the models (see Figure 56 bottom).

VIII. DISTINCTION FROM OTHER SURVEYS

Over the years, many works have been proposed to give an overview of the research carried out and the recent state-of-the-art. However, Deep Learning is currently a hot topic, so research is progressing fast with continuous discoveries and improvements. The same applies to hardware architectures. Although the fundamental blocks are fixed, the paradigms with which they can be combined and exploited are many and varied. Therefore, it is essential to have surveys that periodically collect the newest material and the recent advancements to keep researchers up to date. This is the idea behind this work, which wants to inform hardware designers about the latest architectures and techniques employed in the DL field.

This paper is intended as complementary to the surveys already available in the literature. The authors aim to focus on the hardware architectures for DL that have become available in the last five years, with a cross-cut on the different platforms. Schuman *et al.* [333] have collected and summarized the previous 35 years of discoveries related to neuromorphic computing, with various examples of hardware for neural

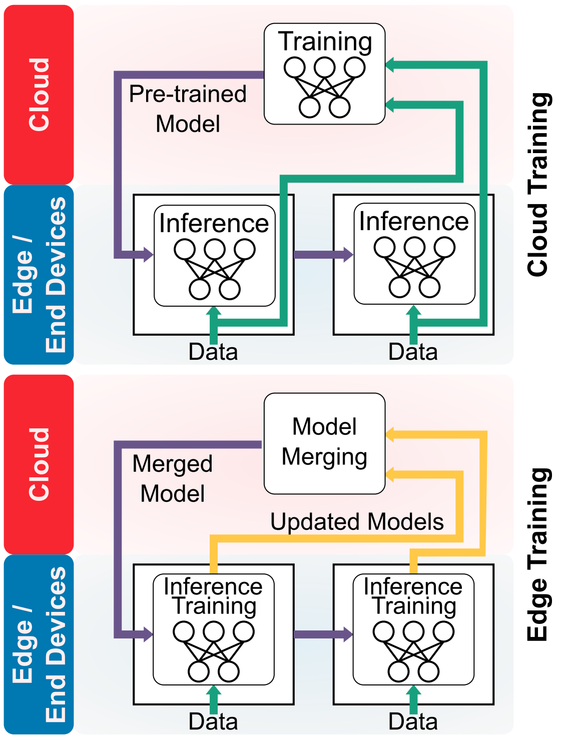


FIGURE 56. Comparison between training performed in the cloud (top) and on edge devices (bottom).

networks. However, the paper does not offer much discussion on the collected material, remaining very compact. On the other hand, Chen *et al.* [334] offer a much broader view, but the examples are limited, and topics such as SNN and adversarial attacks are not covered. Deng *et al.* [335] propose a very complete and extensive work that deals thoroughly with the compression of data taking into account the sparsity and quantization. The work is very comprehensive; nevertheless, it remains very biased toward the compression and lacks of considerations on the SNN. Our work stems from the survey proposed by Sze *et al.* [92], however, updating it with the numerous advances of the last three years and completing it with comprehensive sections on SNNs and adversarial attacks. Table 8 compares a list of state-of-the-art surveys, showing the key aspects that characterize each work.

IX. CONCLUSION

The focus on Deep Learning (DL) has grown exponentially in recent years, as well as the performance of the algorithms and the number of applications that involve it. However, with the increasing complexity of algorithms, the need for hardware devices capable of satisfying the requirements has also increased. The DL has always stood out for its high workload and for being computation-hungry. Moreover, today's trend is to move towards mobile and possibly wearable devices that are part of the IoT whose architectures are heterogeneous, in which general-purpose processors are coupled with dedicated accelerators. The IoT introduces even tighter power constraints considering that many of its nodes are battery-powered or rely on energy harvest systems.

Therefore, it is essential to take into consideration the critical aspects of the hardware already in the design phase. In this regard, there are a large number of techniques to design hardware architectures with high energy-efficiency and high performance without sacrificing accuracy.

This work surveys most of the known techniques to produce energy-efficient dataflows, handling especially the aspects related to memory. The memory hierarchy is deeply analyzed to understand to which levels it is convenient to intervene and, in the ad-hoc architectures, how it must be modelled in order to reduce to the minimum the power consumption. For example, starting from the memory that is the most power-greedy element, it is possible to define a dataflow with a related memory hierarchy that maximizes the data reuse, avoiding continuous access to memory.

The article mainly refers to three models: Deep neural Networks (DNNs), convolutional neural Networks (CNNs) and Spiking Neural Networks (SNNs). While the first two cases are important for the performance and accuracy they have managed to achieve, often beyond the human one, the latter is interesting for the low-power profile and paradigm they represent, considered by many as the third generation of NNs.

In addition to the techniques for developing accelerator architectures, other factors need to be considered like the cybersecurity. In the DL world, security attacks are often represented by noise injection into the input sample to the NN to ensure its misclassification. Many different types of attacks exist, according to the knowledge of the network under attack, the type of perturbation and the target class.

Finally, this work presents which frameworks to use to create or modify models and any datasets to test them on. Benchmarking is a key step in establishing the properties of the networks, but also the hardware on which they are developed. The most critical metrics to define their goodness and comparison with other platforms are examined.

REFERENCES

[1] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, "A comparative study on ASIC, FPGAs, GPUs and general purpose processors in the O(N^2) gravitational n-body simulation," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Jul. 2009, pp. 447–452.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*. Washington, DC, USA: IEEE Computer Society, Dec. 2015, pp. 1026–1034.

[3] D. Zhang and S.-E. Liu, "Top-down saliency object localization based on deep-learned features," in *Proc. 11th Int. Congr. Image Signal Process., Biomed. Eng. Informat. (CISP-BMEI)*, Oct. 2018, pp. 1–9.

[4] T. Treebupachatsakul and S. Poomrittigul, "Bacteria classification using image processing and deep learning," in *Proc. 34th Int. Tech. Conf. Circuits/Syst., Comput. Commun. (ITC-CSCC)*, Jun. 2019, pp. 1–3.

[5] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos, "Image segmentation using deep learning: A survey," *CoRR*, vol. abs/2001.05566, 2020. [Online]. Available: https://arxiv.org/abs/2001.05566

[6] M. Aibin, "Deep learning for cloud resources allocation: Long-short term memory in EONs," in *Proc. 21st Int. Conf. Transparent Opt. Netw. (ICTON)*, Jul. 2019, pp. 1–4.

[7] H. C. Kaskavalci and S. Goren, "A deep learning based distributed smart surveillance architecture using edge and cloud computing," in *Proc. Int. Conf. Deep Learn. Mach. Learn. Emerg. Appl. (Deep-ML)*, Aug. 2019, pp. 1–6.

[8] R. Zanc, T. Cioara, and I. Anghel, "Forecasting financial markets using deep learning," in *Proc. IEEE 15th Int. Conf. Intell. Comput. Commun. Process. (ICCP)*, Sep. 2019, pp. 459–466.

[9] J. J.-C. Ying, P.-Y. Huang, C.-K. Chang, and D.-L. Yang, "A preliminary study on deep learning for predicting social insurance payment behavior," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 1866–1875.

[10] V.-S. Ha, D.-N. Lu, G. S. Choi, H.-N. Nguyen, and B. Yoon, "Improving credit risk prediction in online peer-to-peer (P2P) lending using feature selection with deep learning," in *Proc. 21st Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2019, pp. 511–515.

[11] A. K. Arslan, S. Yasar, and C. Colak, "An intelligent system for the classification of lung cancer based on deep learning strategy," in *Proc. Int. Artif. Intell. Data Process. Symp. (IDAP)*, Sep. 2019, pp. 1–4.

[12] H. Mohsen, E.-S. El-Dahshan, E.-S. El-Horbarty, and A.-B. M. Salem, "Classification using deep learning neural networks for brain tumors," *Future Comput. Informat. J.*, vol. 3, pp. 68–71, Jun. 2017.

[13] C. Barata and J. S. Marques, "Deep learning for skin cancer diagnosis with hierarchical architectures," in *Proc. IEEE 16th Int. Symp. Biomed. Imag. (ISBI)*, Apr. 2019, pp. 841–845.

[14] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *J. Field Robot.*, vol. 37, no. 3, pp. 362–386, Apr. 2020.

[15] T.-H.-S. Li, P.-H. Kuo, C.-Y. Chang, H.-P. Hsu, Y.-C. Chen, and C.-H. Chang, "Deep belief Network–Based learning algorithm for humanoid robot in a pitching game," *IEEE Access*, vol. 7, pp. 165659–165670, 2019.

[16] C. Wang, D. Freer, J. Liu, and G.-Z. Yang, "Vision-based automatic control of a 5-Fingered assistive robotic manipulator for activities of daily living," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Nov. 2019, pp. 627–633.

[17] J. Guan, W. Zhou, S. Kang, Y. Sun, and Z. Liu, "Robot formation control based on Internet of Things technology platform," *IEEE Access*, vol. 8, pp. 96767–96776, 2020.

[18] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "Ultra low power deep-learning-powered autonomous nano drones," *CoRR*, vol. abs/1805.01831, 2018. [Online]. Available: http://arxiv.org/abs/1805.01831

[19] N. Tsang, C. Cao, S. Wu, Z. Yan, A. Yousefi, A. Fred-Ojala, and I. Sidhu, "Autonomous household energy management using deep reinforcement learning," in *Proc. IEEE Int. Conf. Eng., Technol. Innov. (ICE/ITMC)*, Jun. 2019, pp. 1–7.

[20] C. Heghedus, A. Chakravorty, and C. Rong, "Energy load forecasting using deep learning," in *Proc. IEEE Int. Conf. Energy Internet (ICEI)*, May 2018, pp. 146–151.

[21] M. Shafique, T. Theocharides, C.-S. Bouganis, M. A. Hanif, F. Khalid, R. Hafiz, and S. Rehman, "An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 827–832.

[22] A. Marchisio, M. A. Hanif, F. Khalid, G. Plastiras, C. Kyrkou, T. Theocharides, and M. Shafique, "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2019, pp. 553–559.

[23] M. Capra, R. Peloso, G. Masera, M. R. Roch, and M. Martina, "Edge computing: A survey on the hardware requirements in the Internet of Things world," *Future Internet*, vol. 11, no. 4, p. 100, Apr. 2019.

[24] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, vol. 49, Feb. 2014, pp. 269–284.

[25] S. Freeman and H. Hamilton, *Biological Science*. Upper Saddle River, NJ, USA: Prentice-Hall, 2005.

[26] W. Mcculloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, pp. 127–147, Dec. 1943.

[27] F. Rosenblatt, "The perceptron: A perceiving and recognizing automaton (project PARA)," Cornell Aeronaut. Lab., Buffalo, NY, USA, Tech. Rep. 85-460-1, 1957.

[28] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, 2009.

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[30] D. H. Hubel and T. N. Wiesel, "Receptive fields of single neurons in the cat's striate cortex," *J. Physiol.*, vol. 148, no. 3, pp. 574–591, 1959.

[31] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Mach. Learn.*, in Proceedings of Machine Learning Research, vol. 37, F. Bach and D. Blei, Eds. Lille, France, Jul. 2015, pp. 448–456.

[32] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Netw.*, vol. 12, no. 1, pp. 145–151, Jan. 1999.

[33] Y. Nesterov, "A method for unconstrained convex minimization problem with the rate of convergence o(1/k^2)," *Doklady AN USSR*, vol. 269, pp. 543–547, 1983.

[34] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, Feb. 2011.

[35] M. D. Zeiler, "ADADELTA: An adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012. [Online]. Available: http://arxiv.org/abs/1212.5701

[36] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent.*, Dec. 2014, pp. 1–15.

[37] A. Y. Ng, "Feature selection, L_1 vs. L_2 regularization, and rotational invariance," in *Proc. 21st Int. Conf. Mach. Learn. (ICML)*. New York, NY, USA: Association for Computing Machinery, 2004, p. 78.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[39] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 1. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.

[41] 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. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[42] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, San Diego, CA, USA, Y. Bengio and Y. LeCun, Eds., May 2015, pp. 1–14. [Online]. Available: http://arxiv.org/abs/1409.1556

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[45] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 5987–5995.

[46] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 2261–2269.

[47] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS)*. Red Hook, NY, USA: Curran Associates, 2017, pp. 3859–3869.

[48] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 7132–7141.

[49] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.* Salt Lake City, UT, USA: IEEE Computer Society, Jun. 2018, pp. 8697–8710.

[50] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*. Las Vegas, NV, USA: IEEE Computer Society, Jun. 2016, pp. 2818–2826.

[51] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, San Francisco, CA, USA, S. P. Singh and S. Markovitch, Eds., Feb. 2017, pp. 4278–4284.

[52] H. Zhang, C. Wu, Z. Zhang, Y. Zhu, Z. Zhang, H. Lin, Y. Sun, T. He, J. Mueller, R. Manmatha, M. Li, and A. J. Smola, "ResNeSt: Split-attention networks," *CoRR*, vol. abs/2004.08955, 2020. [Online]. Available: https://arxiv.org/abs/2004.08955

[53] T. Ridnik, H. Lawen, A. Noy, and I. Friedman, "TResNet: High performance GPU-dedicated architecture," *CoRR*, vol. abs/2003.13630, 2020. [Online]. Available: https://arxiv.org/abs/2003.13630

[54] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *Proc. 21th Int. Conf. Artif. Neural Netw. (ICANN)*. Berlin, Germany: Springer-Verlag, 2011, pp. 44–51.

[55] G. E. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with EM routing," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, Vancouver, BC, Canada, Apr./May 2018, pp. 1–15.

[56] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A. L. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proc. 15th Eur. Conf. Comput. Vis. (ECCV)*, in Lecture Notes in Computer Science, vol. 11205, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Munich, Germany: Springer, Sep. 2018, pp. 19–35.

[57] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. 36th Int. Conf. Mach. Learn. (ICML)*, in Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., Long Beach, CA, USA, vol. 97, Jun. 2019, pp. 6105–6114.

[58] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Netw.*, vol. 10, no. 9, pp. 1659–1671, Dec. 1997.

[59] N. K. Kasabov, *Time-Space, Spiking Neural Networks and Brain-Inspired Artificial Intelligence*. Berlin, Germany: Springer-Verlag, 2019.

[60] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[61] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker project," *Proc. IEEE*, vol. 102, no. 5, pp. 652–665, May 2014.

[62] S. Schmitt, "Neuromorphic hardware in the loop: Training a deep spiking network on the BrainScaleS wafer-scale system," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, May 2017, pp. 2227–s2234.

[63] M. Davies, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018.

[64] *Building a Silicon Brain*. Accessed: Apr. 23, 2020. [Online]. Available: https://www.the-scientist.com/features/building-a-silicon-brain-65738

[65] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128 × 128 120db 30mw asynchronous vision sensor that responds to relative intensity change," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2006, pp. 2060–2069.

[66] D. Beeman, "Hodgkin-huxley model," in *Encyclopedia of Computational Neuroscience*, D. Jaeger and R. Jung, Eds. New York, NY, USA: Springer, 2013, pp. 1–13.

[67] E. Izhikevich, "Simple model of spiking neurons," *IEEE Trans. Neural Netw.*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.

[68] Z. Wang, L. Guo, and M. Adjouadi, "A generalized leaky integrate-and-fire neuron model with fast implementation method," *Int. J. Neural Syst.*, vol. 24, no. 05, Aug. 2014, Art. no. 1440004.

[69] F. Ponulak and A. Kasiński, "Introduction to spiking neural networks: Information processing, learning and applications," *Acta Neurobiologiae Experimentalis*, vol. 71, no. 4, pp. 409–433, 2011.

[70] B. Ruf and M. Schmitt, "Hebbian learning in networks of spiking neurons using temporal coding," in *Biological and Artificial Computation: From Neuroscience to Technology*, J. Mira, R. Moreno-Díaz, and J. Cabestany, Eds. Berlin, Germany: Springer, 1997, pp. 380–389.

[71] G. Bi and M.-M. Poo, "Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type," *J. Neurosci., Off. J. Soc. Neurosci.*, vol. 18, pp. 10464–10472, Jan. 1999.

[72] G. Srinivasan, P. Panda, and K. Roy, "STDP-based unsupervised feature learning using Convolution-over-time in spiking neural networks for energy-efficient neuromorphic computing," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 14, no. 4, pp. 1–12, Dec. 2018.

[73] S. Fusi, M. Annunziato, D. Badoni, A. Salamon, and D. J. Amit, "Spike-driven synaptic plasticity: Theory, simulation, VLSI implementation," *Neural Comput.*, vol. 12, no. 10, pp. 2227–2258, Oct. 2000.

[74] R. V. W. Putra and M. Shafique, "Fspinn: An optimization framework for memory- and energy-efficient spiking neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3601–3613, Nov. 2020.

[75] B. Rückauer, N. Känzig, S. Liu, T. Delbrück, and Y. Sandamirskaya, "Closing the accuracy gap in an event-based visual recognition task," *CoRR*, vol. abs/1906.08859, 2019. [Online]. Available: http://arxiv.org/abs/1906.08859

[76] S. Bohté, J. Kok, and H. L. Poutré, "SpikeProp: Backpropagation for networks of spiking neurons," in *Proc. ESANN*, 2000, pp. 17–37.

[77] R. Gütig and H. Sompolinsky, "The tempotron: A neuron that learns spike timing–based decisions," *Nature Neurosci.*, vol. 9, no. 3, pp. 420–428, 2006.

[78] R. V. Florian, "The chronotron: A neuron that learns to fire temporally precise spike patterns," *PLoS ONE*, vol. 7, no. 8, Aug. 2012, Art. no. e40233.

[79] F. Ponulak, "Resume-new supervised learning method for spiking neural networks," Tech. Rep., 2005.

[80] A. Mohemmed, S. Schliebs, S. Matsuda, and N. Kasabov, "Span: Spike pattern association neuron for learning spatio-temporal spike patterns," *Int. J. Neural Syst.*, vol. 22, no. 4, Aug. 2012, Art. no. 1250012.

[81] S. M. Bohte, J. N. Kok, and H. La Poutré, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, nos. 1–4, pp. 17–37, Oct. 2002.

[82] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers Neurosci.*, vol. 10, p. 508, Nov. 2016.

[83] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," in *Proc. NIPS*, 2018, pp. 1412–1421.

[84] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," *IEEE Signal Process. Mag.*, vol. 36, no. 6, pp. 51–63, Nov. 2019.

[85] J. C. Thiele, O. Bichler, and A. Dupret, "SpikeGrad: An ANN-equivalent computation model for implementing backpropagation with spikes," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–10.

[86] J. Kaiser, H. Mostafa, and E. Neftci, "Synaptic plasticity dynamics for deep continuous local learning," 2018, *arXiv:1811.10766*. [Online]. Available: https://arxiv.org/abs/1811.10766

[87] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers Neurosci.*, vol. 11, p. 682, Dec. 2017.

[88] R. Massa, A. Marchisio, M. Martina, and M. Shafique, "An efficient spiking neural network for recognizing gestures with a DVS camera on the loihi neuromorphic processor," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2020, pp. 1–10.

[89] N. Rathi, G. Srinivasan, P. Panda, and K. Roy, "Enabling deep spiking neural networks with hybrid conversion and spike timing dependent backpropagation," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–14.

[90] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: Opportunities and challenges," *Frontiers Neurosci.*, vol. 12, p. 774, Oct. 2018.

[91] D. AL-DABASS, P. Vindlacheruvu, and D. J. Evans, "Parallelism in neural nets," *Parallel Algorithms Appl.*, vol. 11, nos. 3–4, pp. 169–185, Jan. 1997.

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

[93] *Intel AVX-512 Instructions*. Accessed: Nov. 27, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html

[94] *BFLOAT16—Hardware Numerics Definition*, Intel, Santa Clara, CA, USA, 2018.

[95] S. L. Gogar. (2017). *BigDL—Scale-Out Deep Learning on Apache Spark* Cluster*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/bigdl-scale-out-deep-learning-on-apache-spark-cluster.html

[96] *NVIDIA Tesla V100 GPU Architecture*, NVIDIA, Santa Clara, CA, USA, 2017.

[97] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Proc. NIPS Workshop Autodiff*, 2017, pp. 1–4.

[98] M. Abadi. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: https://www.tensorflow.org/

[99] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia (MM)*, K. A. Hua, Y. Rui, R. Steinmetz, A. Hanjalic, A. Natsev, and W. Zhu, Eds., Orlando, FL, USA, Nov.2014, pp. 675–678.

[100] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "CuDNN: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[101] *NVIDIA CUDA X*. Accessed: Nov. 26, 2020. [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries

[102] *NVIDIA A100 Tensor Core GPU Architecture*, NVIDIA, Santa Clara, CA, USA, 2020.

[103] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Proc. 10th Int. Workshop Frontiers Handwriting Recognit.*, G. Lorette, Ed. La Baule, France: Université de Rennes 1, Oct. 2006, pp. 1–7. [Online]. Available: http://www.suvisoft.com.

[104] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2017, pp. 19–24.

[105] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, 1969.

[106] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. 24th Int. Conf. Artif. Neural Netw. (ICANN)*, in Lecture Notes in Computer Science, vol. 8681, S. Wermter, C. Weber, W. Duch, T. Honkela, P. D. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, Eds. Hamburg, Germany: Springer, Sep. 2014, pp. 281–290.

[107] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," in *Proc. 2nd Int. Conf. Learn. Represent. (ICLR)*, Y. Bengio and Y. LeCun, Eds., Banff, AB, Canada, Apr. 2014, pp. 1–9.

[108] S. Winograd, *Arithmetic Complexity of Computations* (CBMS-NSF Regional Conference Series in Applied Mathematics). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1980, vol. 33.

[109] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.

[110] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf. (USENIX ATC)*. Berkeley, CA, USA: USENIX Association, 2019, pp. 1025–1040.

[111] A. Rodriguez, E. Segal, E. Meiri, E. Fomenko, Y. Kim, H. Shen, and B. Ziv, "Lower numerical precision deep learning inference and training," Intel, Santa Clara, CA, USA, White Paper, Jan. 2018.

[112] M. Horowitz. *Energy Table for 45 nm Process*. Stanford VLSI Wiki. Accessed: Nov. 26, 2020. [Online]. Available: https://sites.google.com/site/seecproject

[113] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[114] Y.-H. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, Jun. 2017.

[115] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2014, pp. 696–701.

[116] V. Sriram, D. Cox, K. H. Tsoi, and W. Luk, "Towards an embedded biologically-inspired machine vision processor," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2010, pp. 273–278.

[117] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jul. 2009, pp. 53–60.

[118] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. Bell, J. Setter, K. Cao, H. Ha, C. Kozyrakis, and M. Horowitz, "DNN dataflow choice is overrated," *CoRR*, vol. abs/1809.04070, 2018. [Online]. Available: http://arxiv.org/abs/1809.04070

[119] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 247–257.

[120] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H.-J. Yoo, "A1.93TO PS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2015, pp. 1–3.

[121] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 26–35.

[122] N. P. Jouppi, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017.

[123] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic CNN accelerator simulator," *CoRR*, vol. abs/1811.02883, 2019. [Online]. Available: http://arxiv.org/abs/1811.02883

[124] M. A. Hanif, R. V. W. Putra, M. Tanvir, R. Hafiz, S. Rehman, and M. Shafique, "MPNA: A massively-parallel neural array accelerator with dataflow optimization for convolutional neural networks," *CoRR*, vol. abs/1810.12910, 2018. [Online]. Available: http://arxiv.org/abs/1810.12910

[125] C. Luo, Y. Wang, W. Cao, P. H. W. Leong, and L. Wang, "RNA: An accurate residual network accelerator for quantized and reconstructed deep neural networks," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, 2018, pp. 60–603.

[126] A. Marchisio, M. A. Hanif, and M. Shafique, "CapsAcc: An efficient hardware accelerator for CapsuleNets with data reuse," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 964–967.

[127] A. Marchisio and M. Shafique, "CapStore: Energy-efficient design and management of the on-chip memory for capsulenet inference accelerators," *CoRR*, vol. abs/1902.01151, 2019. [Online]. Available: http://arxiv.org/abs/1902.01151

[128] A. Marchisio, V. Mrazek, M. A. Hanif, and M. Shafique, "DESCNet: Developing efficient scratchpad memories for capsule network hardware," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Oct. 13, 2020, doi: 10.1109/TCAD.2020.3030610.

[129] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 92–104.

[130] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W convolutional network accelerator," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 27, no. 11, pp. 2461–2475, Nov. 2017.

[131] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 13–19.

[132] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[133] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 161–170.

[134] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.

[135] S. Yin, P. Ouyang, S. Tang, F. Tu, X. Li, L. Liu, and S. Wei, "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *Proc. Symp. VLSI Circuits*, 2017, pp. C26–C27.

[136] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale DNN processor for real-time AI," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 1–14.

[137] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 461–475.

[138] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 58–70.

[139] H. Liao, J. Tu, J. Xia, and X. Zhou, "DaVinci: A scalable architecture for neural network computing," in *Proc. IEEE Hot Chips 31 Symp. (HCS)*, Aug. 2019, pp. 1–44.

[140] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 29–38.

[141] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A systematic approach to blocking convolutional neural networks," *CoRR*, vol. abs/1606.04209, 2016. [Online]. Available: http://arxiv.org/abs/1606.04209

[142] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 343–348.

[143] L. Ke, X. He, and X. Zhang, "NNest: Early-stage design space exploration tool for neural network inference accelerators," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*. New York, NY, USA: Association for Computing Machinery, 2018.

[144] R. V. W. Putra, M. A. Hanif, and M. Shafique, "ROMANet: Fine-grained reuse-driven data organization and off-chip memory access management for deep neural network accelerators," *CoRR*, vol. abs/1902.10222, 2019. [Online]. Available: http://arxiv.org/abs/1902.10222

[145] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 754–768.

[146] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "MRNA: Enabling efficient mapping space exploration for a reconfiguration neural accelerator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 282–292.

[147] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 304–315.

[148] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "MAGNet: A modular accelerator generator for neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[149] A. Colucci, A. Marchisio, B. Bussolino, V. Mrazek, M. Martina, G. Masera, and M. Shafique, "A fast design space exploration framework for the deep learning accelerators: Work-in-progress," in *Proc. IEEE Int. Conf. Hardw.-Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2020, pp. 34–36.

[150] H. Ahmad, T. Arif, M. A. Hanif, R. Hafiz, and M. Shafique, "SuperSlash: A unified design space exploration and model compression methodology for design of deep learning accelerators with reduced off-chip memory access," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 4191–4204, Nov. 2020.

[151] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–13.

[152] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2815–2823.

[153] S. Zeng, H. Sun, Y. Xing, X. Ning, Y. Shan, X. Chen, Y. Wang, and H. Yang, "Black box search space profiling for accelerator-aware neural architecture search," in *Proc. 25th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2020, pp. 518–523.

[154] A. Marchisio, A. Massa, V. Mrazek, B. Bussolino, M. Martina, and M. Shafique, "NASCaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks," *CoRR*, vol. abs/2008.08476, 2020. [Online]. Available: https://arxiv.org/abs/2008.08476

[155] Q. Lu, W. Jiang, X. Xu, Y. Shi, and J. Hu, "On neural architecture search for resource-constrained hardware platforms," 2019, *arXiv:1911.00105*. [Online]. Available: https://arxiv.org/abs/1911.00105

[156] P. Achararit, M. A. Hanif, R. V. W. Putra, M. Shafique, and Y. Hara-Azumi, "APNAS: Accuracy-and-Performance-Aware neural architecture search for neural hardware accelerators," *IEEE Access*, vol. 8, pp. 165319–165334, 2020.

[157] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Accuracy vs. efficiency: Achieving both through FPGA-implementation aware neural architecture search," in *Proc. 56th Annu. Design Automat. Conf. (DAC)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–6.

[158] W. Jiang, L. Yang, S. Dasgupta, J. Hu, and Y. Shi, "Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start," 2020, *arXiv:2007.09087*. [Online]. Available: https://arxiv.org/abs/2007.09087

[159] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," in *Proc. ICML*, 2018, pp. 1–11.

[160] B. Zoph, D. Yuret, J. May, and K. Knight, "Transfer learning for low-resource neural machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2016, pp. 1568–1575.

[161] D. Stamoulis, R. Ding, D. Wang, D. Lymberopoulos, B. Priyantha, J. Liu, and D. Marculescu, "Single-path NAS: Designing hardware-efficient ConvNets in less than 4 hours," in *Proc. ECML/PKDD*, 2019, pp. 481–497.

[162] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "FBNet: Hardware-aware efficient ConvNet design via differentiable neural architecture search," *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 10726–10734.

[163] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," 2019, *arXiv:1904.00420*. [Online]. Available: http://arxiv.org/abs/1904.00420

[164] L. L. Zhang, Y. Yang, Y. Jiang, W. Zhu, and Y. Liu, "Fast hardware-aware neural architecture search," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2020, pp. 2959–2967.

[165] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Proc. UAI*, 2019, pp. 367–377.

[166] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, Feb. 2014, pp. 10–14.

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

[168] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016. [Online]. Available: http://arxiv.org/abs/1605.04711

[169] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient Integer-Arithmetic-Only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[170] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. Deep Learn. Unsupervised Feature Learn. Workshop NIPS*, 2011, pp. 1–8.

[171] I. Hubara, M. Courbariaux, and D. Soudry, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, pp. 1–30, Jan. 2018.

[172] Y. Guo, "A survey on methods and theories of quantized neural networks," *CoRR*, vol. abs/1808.04752, 2018. [Online]. Available: http://arxiv.org/abs/1808.04752

[173] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 11, pp. 5784–5789, Nov. 2018.

[174] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1MB model size," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[175] C. Sakr and N. Shanbhag, "Per-tensor fixed-point quantization of the back-propagation algorithm," in *Proc. ICLR*, 2019, pp. 1–26.

[176] A. Marchisio, B. Bussolino, A. Colucci, M. Martina, G. Masera, and M. Shafique, "Q-capsnets: A specialized framework for quantizing capsule networks," in *Proc. 57th Annu. Design Automat. Conf.*, Jul. 2020, pp. 1–6.

[177] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 8604–8612.

[178] Y. Umuroglu, L. Rasnayake, and M. Själander, "BISMO: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 307–3077.

[179] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 80–83, Aug. 2017.

[180] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan. 2019.

[181] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Automat. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[182] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 764–775.

[183] S. Ryu, H. Kim, W. Yi, and J. Kim, "BitBlade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *Proc. 56th ACM/IEEE Design Automat. Conf. (DAC)*, Jun. 2019, pp. 1–6.

[184] *NVIDIA Turing GPU Architecture. Graphics Reinvented*, Nvidia, Santa Clara, CA, USA, 2018.

[185] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Computer Vision*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham, Switzerland: Springer, 2016, pp. 525–542.

[186] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[187] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2016, pp. 4107–4115.

[188] 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 Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2016, pp. 236–241.

[189] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, ''Hyperdrive: A multi-chip systolically scalable binary-weight CNN inference engine,'' *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 309–322, Jun. 2019.

[190] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda, and M. Motomura, ''BRein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 TOPS at 0.6 w,'' *IEEE J. Solid-State Circuits*, vol. 53, no. 4, pp. 983–994, Apr. 2018.

[191] F. Conti, P. D. Schiavone, and L. Benini, ''XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference,'' *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2940–2951, Nov. 2018.

[192] A. Al Bahou, G. Karunaratne, R. Andri, L. Cavigelli, and L. Benini, ''XNORBIN: A 95 TOp/s/W hardware accelerator for binary convolutional neural networks,'' in *Proc. IEEE Symp. Low-Power High-Speed Chips (COOL CHIPS)*, Apr. 2018, pp. 1–3.

[193] D. Miyashita, E. H. Lee, and B. Murmann, ''Convolutional neural networks using logarithmic data representation,'' *CoRR*, vol. abs/1603.01025, 2016. [Online]. Available: http://arxiv.org/abs/1603.01025

[194] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, ''LogNet: Energy-efficient neural networks using logarithmic computation,'' in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2017, pp. 5900–5904.

[195] T. Ueki, I. Keisuke, T. Matsubara, and T. Kurokawa, ''AQSS: Accelerator of quantization neural networks with stochastic approach,'' in *Proc. 6th Int. Symp. Comput. Netw. Workshops (CANDARW)*, Nov. 2018, pp. 138–144.

[196] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, ''Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base,'' in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.

[197] Y. Gong, L. Liu, M. Yang, and L. D. Bourdev, ''Compressing deep convolutional networks using vector quantization,'' *CoRR*, vol. abs/1412.6115, 2014. [Online]. Available: http://arxiv.org/abs/1412.6115

[198] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, ''Model compression and acceleration for deep neural networks: The principles, progress, and challenges,'' *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 126–136, Jan. 2018.

[199] Y. LeCun, J. S. Denker, and S. A. Solla, ''Optimal brain damage,'' in *Advances in Neural Information Processing Systems*, D. S. Touretzky, Ed. San Mateo, CA, USA: Morgan Kaufmann, 1990, pp. 598–605.

[200] S. Han, J. Pool, J. Tran, and W. J. Dally, ''Learning both weights and connections for efficient neural networks,'' in *Proc. 28th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 1. Cambridge, MA, USA: MIT Press, 2015, pp. 1135–1143.

[201] S. Srinivas and R. V. Babu, ''Data-free parameter pruning for deep neural networks,'' in *Proc. Brit. Mach. Vis. Conf. (BMVC)*, X. Xie, M. W. Jones, and G. K. L. Tam, Eds., Swansea, U.K., Sep. 2015, pp. 31.1–31.12.

[202] Y. He, X. Zhang, and J. Sun, ''Channel pruning for accelerating very deep neural networks,'' in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 1398–1406.

[203] S. Han, H. Mao, and W. J. Dally, ''Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding,'' in *Proc. 4th Int. Conf. Learn. Represent. (ICLR)*, Y. Bengio and Y. LeCun, Eds., San Juan, Puerto Rico, May 2016, pp.1–14.

[204] A. Marchisio, M. A. Hanif, M. Martina, and M. Shafique, ''PruNet: Class-blind pruning method for deep neural networks,'' in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2018, pp. 1–8.

[205] T.-J. Yang, Y.-H. Chen, and V. Sze, ''Designing energy-efficient convolutional neural networks using energy-aware pruning,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 6071–6079.

[206] H. Yang, Y. Zhu, and J. Liu, ''ECC: Platform-independent energy-constrained deep neural network compression via a bilinear regression model,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 11206–11215.

[207] F. Tung and G. Mori, ''CLIP-Q: Deep network compression learning by in-parallel pruning-quantization,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 7873–7882.

[208] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, ''AMC: AutoML for model compression and acceleration on mobile devices,'' in *Computer Vision*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Cham, Switzerland: Springer, 2018, pp. 815–832.

[209] H. Cai, J. Lin, Y. Lin, Z. Liu, K. Wang, T. Wang, L. Zhu, and S. Han, ''AutoML for architecting efficient and specialized neural networks,'' *IEEE Micro*, vol. 40, no. 1, pp. 75–82, Jan. 2020.

[210] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han, ''APQ: Joint search for network architecture, pruning and quantization policy,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 2075–2084.

[211] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, ''MobileNets: Efficient convolutional neural networks for mobile vision applications,'' *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[212] F. Chollet, ''Xception: Deep learning with depthwise separable convolutions,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 1800–1807.

[213] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky, ''Speeding-up convolutional neural networks using fine-tuned CP-decomposition,'' in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, Y. Bengio and Y. LeCun, Eds., San Diego, CA, USA, May 2015, pp. 1–11.

[214] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, ''Compression of deep convolutional neural networks for fast and low power mobile applications,'' in *Proc. 4th Int. Conf. Learn. Represent. (ICLR)*, Y. Bengio and Y. LeCun, Eds., San Juan, Puerto Rico, May 2016, pp. 1–16.

[215] C. Buciluă, R. Caruana, and A. Niculescu-Mizil, ''Model compression,'' in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 535–541.

[216] L. Ba and R. Caruana, ''Do deep nets really need to be deep?'' in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 3, Jan. 2014, pp. 2654–2662.

[217] G. E. Hinton, O. Vinyals, and J. Dean, ''Distilling the knowledge in a neural network,'' *CoRR*, vol. abs/1503.02531, 2015. [Online]. Available: http://arxiv.org/abs/1503.02531

[218] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, ''FitNets: Hints for thin deep nets,'' in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, Y. Bengio and Y. LeCun, Eds., San Diego, CA, USA, May 2015, pp. 1–13.

[219] J. Yim, D. Joo, J. Bae, and J. Kim, ''A gift from knowledge distillation: Fast optimization, network minimization and transfer learning,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7130–7138.

[220] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, ''Deep mutual learning,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4320–4328.

[221] R. W. Vuduc and J. W. Demmel, ''Automatic performance tuning of sparse matrix kernels,'' Ph.D. dissertation, Graduate Division, Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, 2003.

[222] 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. Comput. Archit. (ISCA)*, Jun. 2016, pp. 1–13.

[223] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, ''SCNN: An accelerator for compressed-sparse convolutional neural networks,'' in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 27–40.

[224] 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,'' in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*. Seoul, South Korea: IEEE Computer Society, Jun. 2016, pp. 243–254.

[225] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, ''NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps,'' *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019.

[226] J. Li, S. Jiang, S. Gong, J. Wu, J. Yan, G. Yan, and X. Li, ''SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules,'' *IEEE Trans. Comput.*, vol. 68, no. 11, pp. 1663–1677, Nov. 2019.

[227] D. Kim, J. Ahn, and S. Yoo, ''ZeNA: Zero-aware neural network accelerator,'' *IEEE Des. Test.*, vol. 35, no. 1, pp. 39–46, Feb. 2018.

[228] Y. Huan, Y. Qin, Y. You, L. Zheng, and Z. Zou, "A low-power accelerator for deep neural networks with enlarged near-zero sparsity," *CoRR*, vol. abs/1705.08009, 2017. [Online]. Available: http://arxiv.org/abs/1705.08009

[229] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Invited–cross-layer approximate computing: From logic to architectures," in *Proc. 53rd Annu. Design Automat. Conf. (DAC)*, 2016, pp. 1–6.

[230] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 1–33 Mar. 2016.

[231] V. Kumar and R. Kant, "Approximate computing for machine learning," in *Proc. 2nd Int. Conf. Commun., Comput. Netw.*, C. R. Krishna, M. Dutta, and R. Kumar, Eds. Singapore: Springer, 2019, pp. 607–613.

[232] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *Proc. 35th Int. Conf. Comput.-Aided Design (ICCAD)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–7.

[233] M. A. Hanif, R. Hafiz, and M. Shafique, "Error resilience analysis for systematically employing approximate computing in convolutional neural networks," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 913–916.

[234] A. Marchisio, V. Mrazek, M. A. Hanif, and M. Shafique, "ReD-CaNe: A systematic methodology for resilience analysis and design of capsule networks under approximations," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1205–1210.

[235] M. A. Hanif, F. Khalid, and M. Shafique, "CANN: Curable approximations for high-performance deep neural network accelerators," in *Proc. 56th Annu. Design Automat. Conf. (DAC)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–6.

[236] X. He, L. Ke, W. Lu, G. Yan, and X. Zhang, "Axtrain: Hardware-oriented neural network training for approximate inference," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–6.

[237] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[238] M. Riaz, R. Hafiz, S. A. Khaliq, M. Faisal, H. T. Iqbal, M. Ali, and M. Shafique, "CAxCNN: Towards the use of canonic sign digit based approximation for hardware-friendly convolutional neural networks," *IEEE Access*, vol. 8, pp. 127014–127021, 2020.

[239] Z. Deng, C. Xu, Q. Cai, P. Faraboschi, and H. Packard, "Reduced-precision memory value approximation for deep learning," Tech. Rep., 2015.

[240] S. Koppula, L. Orosa, A. G. Yağlıkçi, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, "EDEN: Enabling energy-efficient, high-performance deep neural network inference using approximate dram," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 166–181.

[241] C.-Y. Chen, J. Choi, K. Gopalakrishnan, V. Srinivasan, and S. Venkataramani, "Exploiting approximate computing for deep learning acceleration," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 821–826.

[242] M. A. Hanif, A. Marchisio, T. Arif, R. Hafiz, S. Rehman, and M. Shafique, "X-DNNs: Systematic cross-layer approximations for energy-efficient deep neural networks," *J. Low Power Electron.*, vol. 14, no. 4, pp. 520–534, Dec. 2018.

[243] *Alibaba Cloud*. Accessed: Nov. 26, 2020. [Online]. Available: https://alibabacloud.com

[244] *AMAZON*. Accessed: Nov. 26, 2020. [Online]. Available: https://aws.amazon.com

[245] *IBM*. Accessed: Nov. 26, 2020. [Online]. Available: https://www.ibm.com/cloud

[246] *Google Cloud*. Accessed: Nov. 26, 2020. [Online]. Available: https://cloud.google.com

[247] *Colab Research*. Accessed: Nov. 26, 2020. [Online]. Available: https://colab.research.google.com

[248] *Microsoft Azure*. Accessed: Nov. 26, 2020. [Online]. Available: https://azure.microsoft.com

[249] B. Rajendran, A. Sebastian, M. Schmuker, N. Srinivasa, and E. Eleftheriou, "Low-power neuromorphic hardware for signal processing applications: A review of architectural and system-level design approaches," *IEEE Signal Process. Mag.*, vol. 36, no. 6, pp. 97–110, Nov. 2019.

[250] C. Liu, G. Bellec, B. Vogginger, D. Kappel, J. Partzsch, F. Neumärker, S. Höppner, W. Maass, S. B. Furber, R. Legenstein, and C. G. Mayr, "Memory-efficient deep learning on a SpiNNaker 2 prototype," *Frontiers Neurosci.*, vol. 12, p. 840, Nov. 2018.

[251] R. V. Wicaksana Putra, M. Abdullah Hanif, and M. Shafique, "DRMap: A generic DRAM data mapping policy for energy-efficient processing of convolutional neural networks," in *Proc. 57th ACM/IEEE Des. Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[252] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 37–47.

[253] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *Proc. 8th Int. Conf. Learn. Represent. (ICLR)*, Addis Ababa, Ethiopia: OpenReview.net, Apr. 2020. [Online]. Available: https://openreview.net/forum?id=HylxE1HKwS

[254] E. Hoffer, T. Ben-Nun, I. Hubara, N. Giladi, T. Hoefler, and D. Soudry, "Augment your batch: Better training with larger batches," *CoRR*, vol. abs/1901.09335, 2019. [Online]. Available: http://arxiv.org/abs/1901.09335

[255] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on cpus," in *Proc. Annu. Tech. Conf.*, Renton, WA, USA, Jul. 2019, D. Malkhi and D. Tsafrir, Eds. Berkeley, CA, USA: USENIX Association, 2019, pp. 1025–1040.

[256] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm cortex-m cpus," *CoRR*, vol. abs/1801.06601, pp. 1–10, Jan. 2018. [Online]. Available: http://arxiv.org/abs/1801.06601

[257] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors," *Phil. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 378, no. 2164, Dec. 2019, Art. no. 20190155.

[258] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[259] Y. Kim, J. Lee, J.-S. Kim, H. Jei, and H. Roh, "Efficient multi-GPU memory management for deep learning acceleration," in *Proc. IEEE 3rd Int. Workshops Found. Appl. Self Syst.*, Sep. 2018, pp. 37–43.

[260] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Program. Logic Appl.*, Aug. 2009, pp. 32–37.

[261] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management," in *Proc. 56th Annu. Des. Autom. Conf.*, Jun. 2019, pp. 1–6.

[262] J. Park and W. Sung, "FPGA based implementation of deep neural networks using on-chip memory only," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Mar. 2016, pp. 1011–1015.

[263] W. Khwa, J. Chen, J. Li, X. Si, E. Yang, X. Sun, R. Liu, P. Chen, Q. Li, S. Yu, and M. Chang, "A 65nm 4kb algorithm-dependent computing-in-memory sram unit-macro with 2.3ns and 55.8tops/w fully parallel product-sum operation for binary dnn edge processors," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 496–498.

[264] T. Yoo, H. Kim, Q. Chen, T. T. Kim, and B. Kim, "A logic compatible 4t dual embedded dram array for in-memory computation of deep neural networks," in *2019 IEEE/ACM Int. Symp. Low Power Electron. Des. (ISLPED)*, Jul. 2019, pp. 1–6.

[265] S. Angizi, Z. He, D. Reis, X. S. Hu, W. Tsai, S. J. Lin, and D. Fan, "Accelerating deep neural networks in Processing-in-Memory platforms: Analog or digital approach?" in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2019, pp. 197–202.

[266] S. Yin, Z. Jiang, M. Kim, T. Gupta, M. Seok, and J.-S. Seo, "Vesti: Energy-efficient in-memory computing accelerator for deep neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 48–61, Jan. 2020.

[267] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Mach. Learn.*, vol. 81, no. 2, pp. 121–148, 2010.

[268] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *Proc. ICLR*, 2014, pp. 1–10.

[269] M. Shafique, M. Naseer, T. Theocharides, C. Kyrkou, O. Mutlu, L. Orosa, and J. Choi, ''Robust machine learning systems: Challenges,current trends, perspectives, and the road ahead,'' *IEEE Design Test*, vol. 37, no. 2, pp. 30–57, Feb. 2020.

[270] I. Goodfellow, J. Shlens, and C. Szegedy, ''Explaining and harnessing adversarial examples,'' in *Proc. ICLR*, 2015, pp. 1–11.

[271] F. Khalid, M. A. Hanif, S. Rehman, and M. Shafique, ''Security for machine learning-based systems: Attacks and challenges during training and inference,'' in *Proc. Int. Conf. Frontiers Inf. Technol. (FIT)*, Dec. 2018, pp. 327–332.

[272] J. J. Zhang, K. Liu, F. Khalid, M. A. Hanif, S. Rehman, T. Theocharides, A. Artussi, M. Shafique, and S. Garg, ''Building robust machine learning systems: Current progress, research challenges, and opportunities,'' in *Proc. 56th Annu. Design Autom. Conf. 2019*, New York, NY, USA, 2019, pp. 1–4.

[273] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, ''Stealing machine learning models via prediction apis,'' in *Proc. 25th USENIX Conf. Secur. Symp.*, New York, NY, USA, 2016, pp. 601–618.

[274] C. Song, T. Ristenpart, and V. Shmatikov, ''Machine learning models that remember too much,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2017, pp. 587–601.

[275] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, ''Practical black-box attacks against machine learning,'' in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 506–519.

[276] X. Yuan, P. He, Q. Zhu, and X. Li, ''Adversarial examples: Attacks and defenses for deep learning,'' *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 9, pp. 2805–2824, Sep. 2019.

[277] B. Luo, Y. Liu, L. Wei, and Q. Xu, ''Towards imperceptible and robust adversarial example attacks against neural networks,'' in *Proc. AAAI*, 2018, pp. 1–8.

[278] A. Marchisio, G. Nanfa, F. Khalid, M. A. Hanif, M. Martina, and M. Shafique, ''Capsattacks: Robust and imperceptible adversarial attacks on capsule networks,'' *CoRR*, vol. abs/1901.09878, pp. 1–10, Jan. 2019. [Online]. Available: http://arxiv.org/abs/1901.09878

[279] A. Marchisio, G. Nanfa, F. Khalid, M. A. Hanif, M. Martina, and M. Shafique, ''Is spiking secure? A comparative study on the security vulnerabilities of spiking and deep neural networks,'' in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2020, pp. 1–8.

[280] Z. Yahya, M. Hassan, S. Younis, and M. Shafique, ''Probabilistic analysis of targeted attacks using transform-domain adversarial examples,'' *IEEE Access*, vol. 8, pp. 33855–33869, 2020.

[281] B. Biggio, I. Pillai, S. Rota Bulò, D. Ariu, M. Pelillo, and F. Roli, ''Is data clustering in adversarial settings secure?'' in *Proc. ACM Workshop Artif. Intell. Secur.*, New York, NY, USA, 2013, pp. 87–98.

[282] L. Mu noz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli, ''Towards poisoning of deep learning algorithms with back-gradient optimization,'' in *Proc. 10th ACM Workshop Artif. Intell. Secur.*, 2017, pp. 27–38.

[283] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein, ''Poison frogs! Targeted clean-label poisoning attacks on neural networks,'' in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA, 2018, pp. 6106–6116.

[284] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, ''Targeted backdoor attacks on deep learning systems using data poisoning,'' *CoRR*, vol. abs/1712.05526, pp. 1–5, May 2017. [Online]. Available: http://arxiv.org/abs/1712.05526

[285] V. Venceslai, A. Marchisio, I. Alouani, M. Martina, and M. Shafique, ''NeuroAttack: Undermining spiking neural networks security through externally triggered bit-flips,'' in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2020, pp. 1–8.

[286] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Srndic, P. Laskov, G. Giacinto, and F. Roli, ''Evasion attacks against machine learning at test time,'' 2017, *arXiv:1708.06131*. [Online]. Available: http://arxiv.org/abs/1708.06131

[287] A. Kurakin, I. J. Goodfellow, and S. Bengio, ''Adversarial examples in the physical world,'' *CoRR*, vol. abs/1607.02533, pp. 1–5, Dec. 2016. [Online]. Available: http://arxiv.org/abs/1607.02533

[288] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, ''Boosting adversarial attacks with momentum,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Dec. 2018, pp. 9185–9193.

[289] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, ''Towards deep learning models resistant to adversarial attacks,'' in *Proc. 6th Int. Conf. Learn. Represent.*, Vancouver, BC, Canada, Apr./May 2018, pp. 1–5.

[290] F. Khalid, M. A. Hanif, S. Rehman, R. Ahmed, and M. Shafique, ''TrISec: Training data-unaware imperceptible security attacks on deep neural networks,'' in *Proc. IEEE 25th Int. Symp. Line Test. Robust Syst. Des. (IOLTS)*, Jul. 2019, pp. 188–193.

[291] N. Carlini and D. Wagner, ''Towards evaluating the robustness of neural networks,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 39–57.

[292] W. Brendel, J. Rauber, and M. Bethge, ''Decision-based adversarial attacks: Reliable attacks against black-box machine learning models,'' in *6th Int. Conf. Learn. Represent.* Vancouver, BC, Canada, Apr. 2018, pp. 1–12.

[293] T. Brunner, F. Diehl, M. T. Le, and A. Knoll, ''Guessing smart: Biased sampling for efficient black-box adversarial attacks,'' in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 4957–4965.

[294] J. Chen and M. I. Jordan, ''Boundary attack++: Query-efficient decision-based adversarial attack,'' *CoRR*, vol. abs/1904.02144, p. 7, Apr. 2019. [Online]. Available: http://arxiv.org/abs/1904.02144

[295] F. Khalid, H. Ali, M. A. Hanif, S. Rehman, R. Ahmed, and M. Shafique, ''Red-attack: Resource efficient decision based attack for machine learning,'' *CoRR*, vol. abs/1901.10258, pp. 1–4, Jan. 2019. [Online]. Available: http://arxiv.org/abs/1901.10258

[296] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, ''Universal adversarial perturbations,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 86–94.

[297] A. Chakarov, A. V. Nori, S. K. Rajamani, S. Sen, and D. Vijaykeerthy, ''Debugging machine learning tasks,'' *CoRR*, vol. abs/1603.07292, pp. 1–29, Mar. 2016. [Online]. Available: http://arxiv.org/abs/1603.07292

[298] N. Baracaldo, B. Chen, H. Ludwig, and J. A. Safavi, ''Mitigating poisoning attacks on machine learning models: A data provenance based approach,'' in *Proc. 10th ACM Workshop Artif. Intell. Secur.*, New York, NY, USA, 2017, pp. 103–110.

[299] K. Liu, B. Dolan-Gavitt, and S. Garg, ''Fine-pruning: Defending against backdooring attacks on deep neural networks,'' in *Proc. 21st Int. Symp.*, Heraklion, Crete, Greece, Sep. , ser. Lecture Notes in Computer Science, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds., vol. 11050. Springer, 2018, pp. 273–294.

[300] F. Khalid, H. Ali, H. Tariq, M. A. Hanif, S. Rehman, R. Ahmed, and M. Shafique, ''QuSecNets: Quantization-based defense mechanism for securing deep neural network against adversarial attacks,'' in *Proc. IEEE 25th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2019, pp. 182–187.

[301] D. Zhang, T. Zhang, Y. Lu, Z. Zhu, and B. Dong, ''You only propagate once: Accelerating adversarial training via maximal principle,'' in *Proc. NeurIPS*, 2019, pp. 227–238.

[302] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. P. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein, ''Adversarial training for free,'' in *Proc. NeurIPS*, 2019, pp. 3358–3369.

[303] E. Wong, L. Rice, and J. Z. Kolter, ''Fast is better than free: Revisiting adversarial training,'' in *Proc. ICLR*, 2020, pp. 1–5.

[304] F. Khalid, M. A. Hanif, S. Rehman, J. Qadir, and M. Shafique, ''FAdeML: Understanding the impact of pre-processing noise filtering on adversarial machine learning,'' in *Proc. Des., Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 902–907.

[305] J. Cohen, E. Rosenfeld, and Z. Kolter, ''Certified adversarial robustness via randomized smoothing,'' in *Proc. 36th Int. Conf. Mach. Learn.*, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, CA, USA: PMLR, Jun. 2019, pp. 1310–1320.

[306] D. Meng and H. Chen, ''Magnet: A two-pronged defense against adversarial examples,'' in *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2017, pp. 135–147.

[307] X. Wang, R. Hou, B. Zhao, F. Yuan, J. Zhang, D. Meng, and X. Qian, ''Dnnguard: An elastic heterogeneous dnn accelerator architecture against adversarial attacks,'' in *Proc. Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, New York, NY, USA, 2020, pp. 19–34.

[308] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, ''Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,'' *CoRR*, vol. abs/1512.01274, pp. 1–45, Dec. 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[309] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent, ''Chainer: A deep learning framework for accelerating the research cycle,'' in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2019, pp. 2002–2011.

[310] F. Seide and A. Agarwal, ''Cntk: Microsoft's open-source deep-learning toolkit,'' in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, New York, NY, USA, 2016, p. 2135.

[311] Y. Ma, D. Yu, T. Wu, and H. Wang, "Paddlepaddle: An open-source deep learning platform from industrial practice," *Frontiers Data Domputing*, vol. 1, no. 1, p. 105, 2019.

[312] J. Bai. (2019). *Onnx: Open Neural Network Exchange*. [Online]. Available: https://github.com/onnx/onnx

[313] F. Chollet. (2015). *keras*. [Online]. Available: https://github.com/fchollet/keras

[314] A. Krizhevsky, "Learning multiple layers of features from tiny images," Dept. Comput. Sci., University of Toronto, Toronto, CA, USA, Tech. Rep., 2009.

[315] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. 13th Eur. Conf.*, vol. 8693, D. J. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Zurich, Switzerland: Springer, 2014, pp. 740–755.

[316] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, "The open images dataset v4," *Int. J. Comput. Vis.*, vol. 128, no. 7, pp. 1956–1981, Mar. 2020.

[317] V. Lomonaco and D. Maltoni, "Core50: A new dataset and benchmark for continuous object recognition," in *Proc. 1st Annu. Conf. Robot Learn.*, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., vol. 78. New York, NY, USA: PMLR, Nov. 2017, pp. 17–26.

[318] A. Barbu, D. Mayo, J. Alverio, W. Luo, C. Wang, D. Gutfreund, J. Tenenbaum, and B. Katz, "Objectnet: A large-scale bias-controlled dataset for pushing the limits of object recognition models," in *Proc. NeurIPS*, 2019, pp. 9453–9463.

[319] R. Perrault, Y. Shoham, E. Brynjolfsson, J. Clark, J. Etchemendy, B. Grosz, T. Lyons, J. Manyika, S. Mishra, and J. C. Niebles, "The ai index 2019 annual report," AI Index Steering Committee, Human-Centered AI Inst., Stanford Univ., Stanford, CA, USA, Tech. Rep., Dec. 2019.

[320] A. Jadhav and P. Kakade, "Deep learning chip market by chip type (gpu, asic, fpga, cpu, and others), technology (system-on-chip, system-in-package, multi-chip module, and others), and industry vertical (media & advertising, bfsi, it & telecom, retail, healthcare, automotive & transportation, and others) - global opportunity analysis and industry forecast, pp. 2018–2025," Allied Market Research, Pune, Maharashtra, Tech. Rep., Jul. 2018.

[321] Z. Jiang, S. Yin, M. Seok, and J.-S. Seo, "XNOR-SRAM: In-memory computing SRAM macro for Binary/Ternary deep neural networks," in *Proc. IEEE Symp. VLSI Technol.*, Jun. 2018, pp. 173–174.

[322] S. Okumura, M. Yabuuchi, K. Hijioka, and K. Nose, "A ternary based bit scalable, 8.80 TOPS/W CNN accelerator with many-core Processing-in-memory architecture with 896K synapses/mm2," in *Proc. Symp. VLSI Technol.*, Jun. 2019, pp. 1–5.

[323] T. N. Theis and H. P. Wong, "The end of Moore's law: A new beginning for information technology," *Comput. Sci. Eng.*, vol. 19, no. 2, pp. 41–50, 2017.

[324] V. Joshi, M. Le Gallo, S. Haefeli, I. Boybat, S. R. Nandakumar, C. Piveteau, M. Dazzi, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Accurate deep neural network inference using computational phase-change memory," *Nature Commun.*, vol. 11, no. 1, p. 2473, May 2020.

[325] S. R. Nandakumar, I. Boybat, V. Joshi, C. Piveteau, M. Le Gallo, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Phase-change memory models for deep learning training and inference," in *Proc. 26th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Nov. 2019, pp. 727–730.

[326] H. Yan, H. R. Cherian, E. C. Ahn, X. Qian, and L. Duan, "ICELIA: A full-stack framework for STT-MRAM-Based deep learning acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 408–422, Feb. 2020.

[327] S. Chattopadhyay, K. Brahma, A. Ray, and M. Sharad, "STT-MRAM for low power access for read-intensive parallel deep-learning architectures," in *Proc. IEEE Int. Symp. Nanoelectron. Inf. Syst.*, 2017, pp. 61–65.

[328] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 541–552.

[329] A. Mutiara, "IEEE computer society's top 12 technology trends for 2020," IEEE Comput. Soc., Washington, DC, USA, Tech. Rep., Dec. 2019.

[330] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, 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. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 14–26.

[331] Y. Kim, H. Kim, D. Ahn, and J.-J. Kim, *Input-Splitting of Large Neural Networks for Power-Efficient Accelerator With Resistive Crossbar Memory Array*. New York, NY, USA: Association for Computing Machinery, 2018.

[332] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann, "An always-on 3.8μj/86% cifar-10 mixed-signal binary cnn processor with all memory on chip in 28nm cmos," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, 2018, pp. 222–224.

[333] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware," *CoRR*, vol. abs/1705.06963, pp. 1–4, Dec. 2017. [Online]. Available: http://arxiv.org/abs/1705.06963

[334] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, Mar. 2020.

[335] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[336] A. A. Ratnaparkhi, E. Pilli, and R. C. Joshi, "Survey of scaling platforms for deep neural networks," in *Proc. Int. Conf. Emerg. Trends Commun. Technol. (ETCT)*, Nov. 2016, pp. 1–6.

[337] C. Zhang and W. Xu, "Neural networks: Efficient implementations and applications," in *Proc. IEEE 12th Int. Conf. (ASICON)*, Oct. 2017, pp. 1029–1032.

[338] M. Kotlar, D. Bojic, M. Punt, and V. Milutinovic, "A survey of deep neural networks: Deployment location and underlying hardware," in *Proc. 14th Symp. Neural Netw. Appl. (NEUREL)*, Nov. 2018, pp. 1–6.

[339] A.-A. Erofei, C.-F. Druta, and C. Daniel Caleanu, "Embedded solutions for deep neural networks implementation," in *Proc. IEEE 12th Int. Symp. Appl. Comput. Intell. Informat. (SACI)*, May 2018, pp. 000425–000430.

[340] A. Sebastian, I. Boybat, M. Dazzi, I. Giannopoulos, V. Jonnalagadda, V. Joshi, G. Karunaratne, B. Kersting, R. Khaddam-Aljameh, S. R. Nandakumar, A. Petropoulos, C. Piveteau, T. Antonakopoulos, B. Rajendran, M. L. Gallo, and E. Eleftheriou, "Computational memory-based inference and training of deep neural networks," in *Proc. Symp. VLSI Technol.*, Jun. 2019, pp. 168–169.

[341] C. Ababei and M. G. Moghaddam, "A survey of prediction and classification techniques in multicore processor systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1184–1200, May 2019.

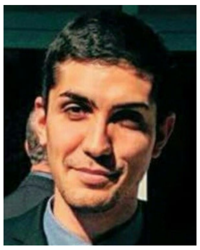
[342] S. Sun, Z. Cao, H. Zhu, and J. Zhao, "A survey of optimization methods from a machine learning perspective," *IEEE Trans. Cybern.*, vol. 50, no. 8, pp. 3668–3681, Aug. 2019.

[343] M. Alam, M. D. Samad, L. Vidyaratne, A. Glandon, and K. M. Iftekharuddin, "Survey on deep neural networks in speech and vision systems," *CoRR*, vol. abs/1908.07656, 2019. [Online]. Available: http://arxiv.org/abs/1908.07656

[344] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artif. Intell. Rev.*, vol. 53, no. 8, pp. 5455–5516, Apr. 2020.

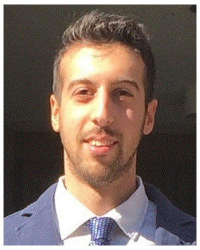
[345] H. Yingge, I. Ali, and K.-Y. Lee, "Deep neural networks on chip—A survey," in *Proc. IEEE Int. Conf. Big Data Smart Comput.*, Feb. 2020, pp. 589–592.

[346] Z. Li, W. Yang, S. Peng, and F. Liu, "A survey of convolutional neural networks: Analysis, applications, and prospects," *CoRR*, vol. abs/2004.02806, 2020. [Online]. Available: https://arxiv.org/abs/2004.02806

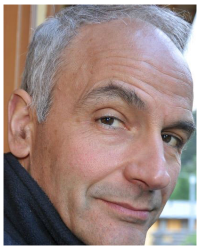


MAURIZIO CAPRA (Graduate Student Member, IEEE) received the bachelor's degree in electronic engineering and the master's degree in electronic engineering, with career in electronic systems, from the Politecnico di Torino, Turin, Italy, in October 2015 and April 2018, respectively, where he is currently pursuing the Ph.D. degree in electronics and communication engineering, under the supervision of Prof. M. Martina. His research interests include newly dedicated architectures for machine learning with emphasis on on-chip learning based on a vertical approach: starting from the algorithm (top) going to the physical implementation (bottom). He is a part of the board of the IEEE Student Branch at the Politecnico di Torino.

BEATRICE BUSSOLINO (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering from the Politecnico di Torino, Turin, Italy, in October 2017 and October 2019, respectively, where she is currently pursuing the Ph.D. degree in electrical, electronics, and communications engineering, under the supervision of Prof. M. Martina. Her current research interests include machine learning and deep neural networks (DNNs) in particular. The focus of her research activity is the development of on-chip architectures for the edge deployment of DNNs. In 2020, she received the Richard Newton Young Fellow Award and won the DAC Young Fellow Poster Presentation Award.



ALBERTO MARCHISIO (Graduate Student Member, IEEE) received the B.Sc. degree in electronic engineering and the M.Sc. degree in electronic engineering (electronic systems) from the Politecnico di Torino, Turin, Italy, in October 2015 and April 2018. He is currently pursuing the Ph.D. degree with the Computer Architecture and Robust Energy-Efficient Technologies (CARE-Tech.) Laboratory, Institute of Computer Engineering, Technische Universität Wien (TU Wien), Vienna, Austria, under the supervision of Dr. M. Shafique. His main research interests include hardware and software optimizations for machine learning, brain-inspired computing, VLSI architecture design, emerging computing technologies, robust design, and approximate computing for energy efficiency. He received the honorable mention at the Italian National Finals of Maths Olympic Games in 2012, and the Richard Newton Young Fellow Award in 2019.



GUIDO MASERA (Senior Member, IEEE) received the Dr.-Ing. *(summa cum laude)* and Ph.D. degrees in electronic engineering from the Politecnico di Torino, Italy, in 1986 and 1992, respectively. He is currently a Professor with the Electronic Department, Politecnico di Torino, since 1992. His research interests include several aspects in the design of digital integrated circuits and systems, with a special emphasis on high-performance architectures for communications, forward error correction, image and video coding, cryptography, and hardware accelerators for machine learning. He has more than 200 publications in the fields of VLSI design and communications. He is an Associate Editor of the IEEE Transactions on Circuits and Systems—I: Regular Papers and *Electronics* (MDPI) and a former Associate Editor of the IEEE Transactions on Circuits and Systems—II: Express Briefs and the *IET Circuits, Devices, and Systems*.

MAURIZIO MARTINA (Senior Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from the Politecnico di Torino, Italy, in 2000 and 2004, respectively. He is currently an Associate Professor from the VLSI-Lab Group, Politecnico di Torino. His research interests include VLSI design and implementation of architectures for digital signal processing, video coding, communications, artificial intelligence, machine learning, and event-based processing. He edited one book and published three book chapters on VLSI architectures and digital circuits for video coding, wireless communications, and error correcting codes. He has more than 100 scientific publications and is author of two patents. He is currently an Associate Editor of the IEEE Transactions on Circuits and Systems—I: Regular Papers. He had been part of the Organizing and a Technical Committee of several international conferences, including BioCAS 2017, ICECS 2019, and AICAS 2020. He is also the Counselor of the IEEE Student Branch at the Politecnico di Torino and a Professional Member of IEEE HKN.

MUHAMMAD SHAFIQUE (Senior Member, IEEE) received the Ph.D. degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in 2011.

Afterwards, he established and led a highly recognized research group, KIT, for several years as well as conducted impactful Research and Development activities in Pakistan. In October 2016, he joined the Institute of Computer Engineering, Faculty of Informatics, Technische Universität Wien (TU Wien), Vienna, Austria, as a Full Professor of Computer Architecture and Robust, Energy-Efficient Technologies. Since September 2020, he has been with the Division of Engineering, New York University, Abu Dhabi (NYU AD), United Arab Emirates. He is currently a Global Network Faculty with the NYU Tandon School of Engineering, USA. His research interests include brain-inspired computing, AI & machine learning hardware and system-level design, energy-efficient systems, robust computing, hardware security, emerging technologies, FPGAs, MPSoCs, and embedded systems. His research has a special focus on cross-layer analysis, modeling, design, and optimization of computing and memory systems. The researched technologies and tools are deployed in application use cases from the Internet-of-Things (IoT), smart cyber-physical systems (CPS), and ICT for development (ICT4D) domains.

Dr. Shafique has given several Keynotes, Invited Talks, and Tutorials, as well as organized many special sessions at premier venues. He has served as the PC Chair, the General Chair, the Track Chair, and the PC member for several prestigious IEEE/ACM conferences. He holds one U.S. patent has (co-) authored six Books, more than ten Book Chapters, and more than 250 articles in premier journals and conferences. He received the 2015 ACM/SIGDA Outstanding New Faculty Award, the AI 2000 Chip Technology Most Influential Scholar Award in 2020, six gold medals, and several best paper awards and nominations at prestigious conferences.