

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering
(34^{th}cycle)

Application Specific Domain Co-design Hardware Accelerator IP for Deep Learning Enabled Internet-of-Things

By

Maurizio Capra

******

Supervisor(s):
Prof. Maurizio Martina, Supervisor

Doctoral Examination Committee:
Prof. Giansalvo Cirrincione , Referee, University of Picardie Jules Verne
Prof. Maurizio Valle, Referee, University of Genova
Prof. Guido Masera, Politecnico di Torino
Prof. Mario Casu, Politecnico di Torino
Dr. Riccardo Peloso, ST Microelectronics

Politecnico di Torino
2022

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Maurizio Capra
2022

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Alla mia famiglia,
a Davide

Abstract

Currently, DeepLearning (DL) is becoming ubiquitous in everyday life. In particular, Convolutional Neural Networks (CNNs) are already present in many applications related to computer vision ranging from medicine to autonomous driving, neural language processing, and finance. However, these algorithms employ very deep networks to achieve impressive performance, 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.

The growing desire to adopt these algorithms in an increasing number of applications and the evolution of hardware devices is recently shifting the inference process from the cloud to the edge, i.e., on embedded systems with limited resources. In addition, this paradigm shift can offer the user lower latency and greater privacy. In-device CNN processing presents several benefits.

Since the data does not have to be transmitted to a central processing unit, it remains local, ensuring better user privacy and more deterministic latency. In addition, the absence of transmission reduces power consumption and the need for a network infrastructure capable of handling large amounts of data. At the same time, however, edge computing demands addressing the gap between the computational and memory-hungry CNNs requirements and limited hardware and energy-constrained embedded platforms. Consequently, the need for cost-effective hardware platforms and co-design optimization techniques capable of implementing energy-efficient DL execution arises.

Achieving the objective of enabling minimal resource systems for CNNs algorithms relies on the ability to optimize and co-design accelerator architectures. This dissertation proposes an in-depth analysis and design development for the acceleration of CNN algorithms for edge computing. The analysis covers all critical

aspects of acceleration, from algorithmic optimization to architectural exploration and hardware architecture development. As widely discussed, IoT, or edge computing, imposes stringent constraints on area, throughput, and energy efficiency. In order to optimize a hardware architecture on these three key metrics, the focus must be placed on a specific application. Identifying the specific application domain enables software-hardware co-design techniques to maximize performance.

This thesis is mainly divided into two parts, each proposing a domain-specific solution, the first aimed at training and the second at inference. The first part concerns the optimization of the training and generalization phase in the context of continuous learning (CL). CL requires the ability to learn from a continuous stream of data that may also include new categories of objects. Extreme Learning Machine (ELM), an alternative to Backpropagation, is adapted and applied to CL to solve the challenges presented by the latter, such as latency, accuracy, and resources (memory, algorithmic complexity, etc.). This approach shows encouraging results, for instance, in the case of CaffeNet it out-performed the current state-of-the-art methodologies, reaching an accuracy of 47%.

The second example introduces two implementations of the Serial-MAC-engine (SMAC-engine), a fully-digital hardware accelerator for inference of quantized CNNs suitable for integration in a heterogeneous System-on-Chip (SoC). With scalable performance, the SMAC engine supports configurable precision for weights (8/6/4 bits) and activations (8/4 bits). Results in 65 nm technology demonstrate that the serial-MAC approach enables the accelerator to achieve a maximum throughput of 14.28 GMAC/s, consuming 0.58 pJ/MAC @ 1.0 V when operating at a precision of 4 bits for weights and 8 bits for activations. The architecture is then further developed to take advantage of the sparsity of activations to improve performance, reaching peaks of 56.88 GMAC/s for Pa=8 and Pw=4 and sparsity 80%.

Contents

List of Figures **x**

List of Tables **xv**

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Neural Network basics 5

1.2.1 Layers 6

1.2.2 Training vs Inference 12

1.2.3 DNN Models 14

1.3 Contributions and Organization 19

2 Background: Architecture Exploration **24**

2.1 Introduction 24

2.2 Temporal vs Spatial architectures 24

2.2.1 Temporal architectures and software optimizations 25

2.2.2 Spatial architectures and dataflow processing 28

2.3 Full precision vs quantized implementations 39

2.4 Activations and Weights Sparsity: Strategies and Encoding 43

2.5 Conclusions 47

3 Enabling the Training on the Edge: an alternative to Backpropagation 48

3.1 Introduction . . . 48

3.2 Continual Learning (CL) paradigm, the road towards on-chip learning 50

3.2.1 Background . . . 50

3.2.2 Benchmarking . . . 53

3.3 Extreme Learning (ELM), an alternative to classic Back-propagation 54

3.3.1 Background . . . 54

3.3.2 Online Sequential-Extreme Learning Machine (OS-ELM): ELM with batches . . . 57

3.4 Contribution and key idea: ELM for Single Incremental Task scenarios 59

3.4.1 ELM and OS-ELM: evaluation and comparison with Back-propagation . . . 59

3.4.2 Key Idea: OS-ELM for SIT . . . 63

3.4.3 Experimental Set-up . . . 64

3.4.4 Accuracy Evaluation . . . 65

3.4.5 Performance Evaluation . . . 66

3.5 Conclusions . . . 68

4 A Multi-Precision Bit-Serial Hardware Accelerator IP for IoT 69

4.1 Architecture . . . 71

4.1.1 SMAC . . . 72

4.1.2 SMAC-engine . . . 75

4.1.3 PULPissimo-integration . . . 76

4.1.4 Performance and considerations . . . 78

4.2 Enhancing Performance with Sparsity . . . 81

4.2.1 Activation Sparsity Distribution Analysis . . . 81

4.2.2 Data Compression and Dataflow . . . 83

4.2.3 Sparse SMAC engine 86
4.2.4 Performance and considerations 89
4.2.5 Conclusions 92

5 Functional Verification beyond ISA architectures. 94
5.1 Beyond ISA Architectures, Challenges, and Future Trends 95
5.2 Challenges 96
5.2.1 Reducing verification time 96
5.2.2 Limited tool scalability 97
5.2.3 Specification drafting 97
5.2.4 Use case identification 98
5.2.5 Security and functional safety 98
5.3 Verification trends 98
5.4 Simulation UVM-based Testbench: basic structure 100
5.4.1 Test 102
5.4.2 Environment 102
5.4.3 Scoreboard 102
5.4.4 Agent 103
5.4.5 Sequencer and sequence 103
5.4.6 Driver 104
5.4.7 Monitor 104
5.4.8 DUT 104
5.5 Conclusion 104

6 Conclusions and Future Works 106
6.1 Future Perspectives 108

References 110

Appendix A List of Publications 122

List of Figures

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

1.2 Model of a biological neuron, adapted from [2] [1]. 5

1.3 Model of an artificial neuron. [1] 6

1.4 Examples of **(a)** a feedforward NN and **(b)** a recurrent NN. [1] . . . 6

1.5 Pseudocode of an FC layer. [1] 7

1.6 Example of an FC layer (left) and of how it can be modeled by a vector-matrix multiplication (right). [1] 7

1.7 Pseudocode of a Conv layer. [1] 9

1.8 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**. [1] 9

1.9 Examples of MaxPooling (left) and AveragePooling (right) layers. [1] 10

1.10 Features, achievable tasks and applications of the three existing ways of training (supervised, unsupervised, and reinforcement) [1] 13

1.11 Timeline of significant neural networks models with the accuracies achieved on the ImageNet dataset and the number of parameters. [1] 15

1.12 Inception Module used in GoogLeNet [3]. Convolutions are performed in four parallel branches and the four outputs are concatenated to produce the single output of the Inception module. [1] . . . 17

1.13 Skip connection modules used in Residual Networks [4]. 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. [1] 17

1.14 Dense blocks used in DenseNets [5]. The output and input of each blue block, that is the series of two convolutions, are concatenated. [1] 18

1.15 Residual module as modified in Squeeze-and-Excitation Networks [6]. 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. [1] 18

1.16 Capsule Network [7]. [1] 19

1.17 Thesis overview 20

2.1 Basic models of temporal **(left)** and spatial **(right)** architectures. [1] 25

2.2 GFLOPS comparison between different CPUs and GPUs. [1] 26

2.3 GFLOPS/WATT comparison between different CPUs and GPUs. [1] 26

2.4 Discontinuity in memory accesses when performing convolution on a matrix stored by rows. [1] 28

2.5 Convolution lowering: mapping a convolution to a matrix-matrix multiplication by rearranging the matrices. [1] 29

2.6 Energy breakdown of two state-of-the-art DNNs accelerators. [1] . . 30

2.7 **(left)** General structure of a hardware accelerator for DNNs and **(right)** interconnection schemes. [1] 31

2.8 Data reuse in an FC layer and in a Conv layer. [1] 32

2.9 Spatial and temporal mapping of the Multiply-and-Accumulate (MACs) operations to the Processing Elements (PEs). [1] 33

2.10 Spatial mapping of the operations in a weight stationary dataflow. [1] 33

2.11 **(top)** PEs array and **(bottom)** temporal-spatial mapping of the operations in a weight stationary accelerator with inputs broadcasting and outputs forwarding. [1] 34

2.12 **(top)** PEs array and **(bottom)** temporal-spatial mapping of the operations in a weight stationary accelerator with inputs and outputs forwarding. [1] 35

2.13 Loop reordering of the 7-nested loops representation of the Conv layer. [1] 36

2.14 **(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. [1] . 37

2.15 **(left)** Spatial mapping of the operations in an output stationary dataflow. **(right)** Modified PE for an output stationary accelerator. [1] 38

2.16 **(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. [1] 38

2.17 Different solutions [8][9][10] to spatially unroll the computations in an output stationary dataflow. [1] 39

2.18 Operations mapping in a row stationary dataflow. [1] 39

2.19 No local reuse dataflow. [1] 40

2.20 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. [1] 41

2.21 **(top)** 32-bit floating-point representation. **(bottom)** 8-bit dynamic fixed-point representation. [1] 41

2.22 Linear, logarithmic and vector quantization techniques. [1] 42

2.23 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). [1] 44

2.24 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 ($3 \times 3 \times 384$) with data parallelism of 8 bits, while the lower bound is based on Conv 1 ($11 \times 11 \times 3$) represented on 32 bits. [1] 46

3.1 Comparison between Multi-Task and Single-incremental-task scenarios for both training and evaluation stages. 52

3.2 Some samples from CORe50 [11]. CORe50 is a dataset consisting of 50 household objects divided into 10 different categories such as: balls, cups, cell phones, light bulbs, scissors, etc. Each object has multiple samples from different angles. 54

3.3 ELM training performance on MINIST split (left column) and CIFAR10 split (right column). 60

3.4 P-ELM vs. OS-ELM comparison in terms of time per batch size. . . 63

3.5 Results comparison for CL NC performed on CaffeNet (left) and GoogLeNet (right). 66

4.1 Input/Output stationary data flow visualization. [12] 73

4.2 Datapath architecture of a single SMAC block.[12] 74

4.3 Serial vs. paralllel approach for a single SMAC block. 75

4.4 HWPE architecture. The SMAC-engine is integrate in PULPissimo by means of a control and a Streamer. The first orchestrates the elaboration thanks to a Low-Level and a High-Level FSM and a register file, while the second manages the data streams.[12] 77

4.5 Layers throughput over several bitwidth combinations.[12] 79

4.6 Sparsity Analysis for ResNet 18 (left), average zero sequences in channel direction (right). 82

4.7 Sparsity Analysis for ResNet 34 (left), average zero sequences in channel direction (right). 82
4.8 Available bandwidth towards TCDM and activation distribution fetch. 84
4.9 Dataflow for sparse SMAC-engine. 85
4.10 Sparse SMAC-block. 86
4.11 SMAC architecture for sparse convolution. 88
4.12 Third accumulation stage in sparse architecture. 89
4.13 VGG16 throughput w.r.t. all the bitwidth combinations with sparsity 0%. 90
4.14 VGG16 Conv8 sparsity analysis. 91

5.1 Design vs. verification effort(left), verification breakdown (right). . 99
5.2 UVM-based testbench architecture. 101
5.3 Scoreboard and golden model. 102
5.4 Testbench agent, composed of monitors, drivers and sequencers. . . 103
5.5 Design Under Test. 104

List of Tables

1.1 Comparison of the models presented in Section 1.2.2 16

1.2 List of Acronyms. 23

3.1 MT and SIT current becnhmarking scenario. 53

3.2 Training latency for BP and ELM on different devices. 61

3.3 OS-ELM training test of a SFLN on CIFAR10 with different batch sizes. 62

3.4 Summary of network adjustments to accomodate OS-ELM. 64

3.5 ELM computation complexity. 67

4.1 Comparative table.[12] 78

4.2 Number of cycle and power consumption with different data bitwidth related to a filter of size $3 \times 3 \times 128 \times 128$.[12] 80

4.3 Comparison between base implementation and sparse implementation for Pa=8 and Pw=4 and sparsity 0% 91

Chapter 1

Introduction

1.1 Context and Motivation

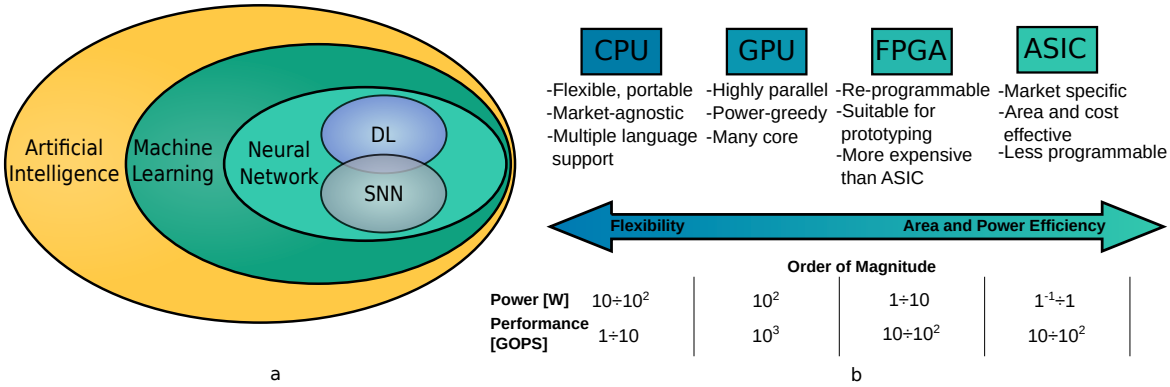


Fig. 1.1 a) Artificial intelligence overview. b) Hardware platforms comparison [1].

Artificial intelligence (AI) represents an essential pillar in many applications and systems. AI is altering the way we interact with technology to the point that we continually employ it without realizing it. Numerous techniques fall under the umbrella of AI, but one, in particular, stands out among all techniques, namely Machine Learning (ML). ML is extensively employed in various application domains, taking advantage of a broad spectrum of flexible and easy-to-learn statistical patterns. ML, in turn, is further composed of several sub-topics, as shown in Figure 1.1 (a). The most popular ones are the 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 outstanding accuracy, exceeding the human one in some tasks, e.g., in image classification [13]. Moreover, it can extract high-level attributes from many raw features coming from sensors. Two factors enable this high accuracy: (1) the computational capability of the latest generation processors and (2) the massive amount of available training data. The latter allows DL to learn different patterns and derive specific predictions using deeper and more complex models. Generally speaking, the larger the training dataset, the better the DL algorithms can learn.

DL-based applications are numerous and cover a wide range of subjects such as:

- Computer Vision: used to extract meaningful features from video and pictures. It includes object localization [14], image classification [15], and image segmentation [16]. Their use is valuable for controlling web traffic [17] or for example, video surveillance [18].
- Business and Finance: deployed to forecast market behavior [19], including insurance [20] and lending [21].
- Healthcare: widely used as aid system in cancer detection such as lung cancer [22], brain cancer [23], skin cancer [24], and many others are continuously rising.
- Robotics: DNNs served in a wide range of use cases like autonomous vehicles [25], humanoid robots [26], assistive robots [27], swarms [28], and drone control system [29].
- Smart Energy Management: used to manage [30] and forecaste [31] the required amount of energy consumption.

Since models are composed of millions of trainable parameters and the datasets are constantly growing, the training phase takes a long time (hours or days according to the application and desired accuracy). Therefore, there is an increasing desire for hardware platforms capable of providing ready-to-use models within a reasonable time. On the other hand, new software learning strategies are needed to try to relieve some of the stress from the hardware side and, in specific cases, propose ad-hoc solutions with a good trade-off between accuracy and complexity.

Similarly, the inference phase feels the need to speed up the execution of DNNs algorithms. In particular, CNNs and RNNs, which represent state-of-the-art applications in a broad range, demand more and more hardware support.

Both inference and training processes require performing tens of billions of multiplication and accumulation operations (MACs) that make these models extremely computationally intensive. In addition, each MAC may require up to 3 memory accesses, two for input elements and one to save the result. Consequently, running these algorithms with minimal latency creates an additional critical constraint on memory bandwidth.

For the previously stated reasons, in multiple cases, CPUs cannot meet the computation requirements, therefore, GPUs are one of the most attractive options for executing such complex models. However, the current trend is driven by Internet-of-Things (IoT) applications that demand more computation capability close to the sensors. This approach of relocating resources toward the IoT nodes is also known as edge computing. This has become feasible for two reasons. Firstly the cost per silicon area has fallen, allowing large-scale production of devices to embed in IoT nodes. Second, by performing on-site operations, there is no longer a need to transfer data to a central server; thus, the distribution of computing capability reduces both latency and the large amounts of energy required for transmission, as well as preserving the privacy of edge nodes' data.

The mesh of IoT nodes is subjected to stringent power constraints. Indeed, most of the nodes are battery-powered or rely on energy harvesting systems [32]. Consequently, integrating a high-end GPU into such a system is unfeasible since the required power would exceed the power envelope of the IoT-edge platforms.

In this scenario, DL algorithms need to be accelerated by the mean of flexible low-power FPGAs that can be reprogrammed, or specialized accelerators in the form of highly optimized ASIC-IPs tailored for the specific application. The recent trend of integrated systems is moving toward heterogeneous multicore systems [33], which feature a mix of general-purpose cores and specialized low-power hardware accelerators. The flexibility of FPGA and ASIC designs (Figure 1.1b) allows a wide range of possible hardware optimizations that are critical for an energy-efficient acceleration of DL models.

This thesis work proposes a deep analysis of the acceleration of DL algorithms for edge computing. This analysis attempts to cover all critical aspects of acceleration,

ranging from architectural exploration through algorithmic optimization and ending with the development of the actual hardware architecture. As mentioned before, IoT, or edge computing, imposes stringent constraints regarding the area, throughput, and power efficiency. The best way to optimize a hardware architecture on these three fundamental metrics is typically to focus on a specific application. Once the application-specific domain is identified, software-hardware co-design techniques can be adopted to maximize performance.

In order to be able to optimize algorithms and hardware for given targets, it is essential to have a good understanding of architectural choices in terms of both NN model and design. For this reason, an architectural exploration is essential. For example, the choice of dataflow is reflected in the spatial topology of the design, just as quantization is reflected in the throughput and parallelism of the architecture as well as the accuracy. Finally, the adoption of specific compression techniques or strategies that leverage a particular data property (e.g., sparsity) can benefit the acceleration of the algorithm.

This dissertation will give two examples of application-specific domain solutions, one aimed at training and one at inference. The former concerns the optimization of the training and generalization phase in the context of Continual Learning (CL). CL requires the ability to learn from a continuous data stream that may also include new tasks (i.e. new object categories in case of object recognition). Multiple challenges arise in this context in terms of latency, accuracy and resources (memory, algorithmic complexity, etc.). The second example, on the other hand, concerns the optimization of the inference phase in the context of CNNs deployed on heterogeneous low-power platforms. In this context, developing two serial-based architectures with an incremental approach allows us to maximize the GMAC/s/W ratio and achieve outstanding performance for IoT node deployment.

When architectures are optimized and taken to extremes, they often require the interaction of complex blocks. Verification of the integrity and operation of such blocks requires as much effort as the design phase. Consequently, there is a need for robust and effective verification techniques.

The following sections introduce a deep background of the most successful Deep Learning models and a careful explanation of inference and training. Then, finally, a section that will describe the contribution of this dissertation and an overview of the upcoming chapters.

1.2 Neural Network basics

The constituent building block of a neural network is the *neuron*, namely *perceptron*, a computational element that attempts to model the comportment of a biological neuron, as shown in Figure 1.2.

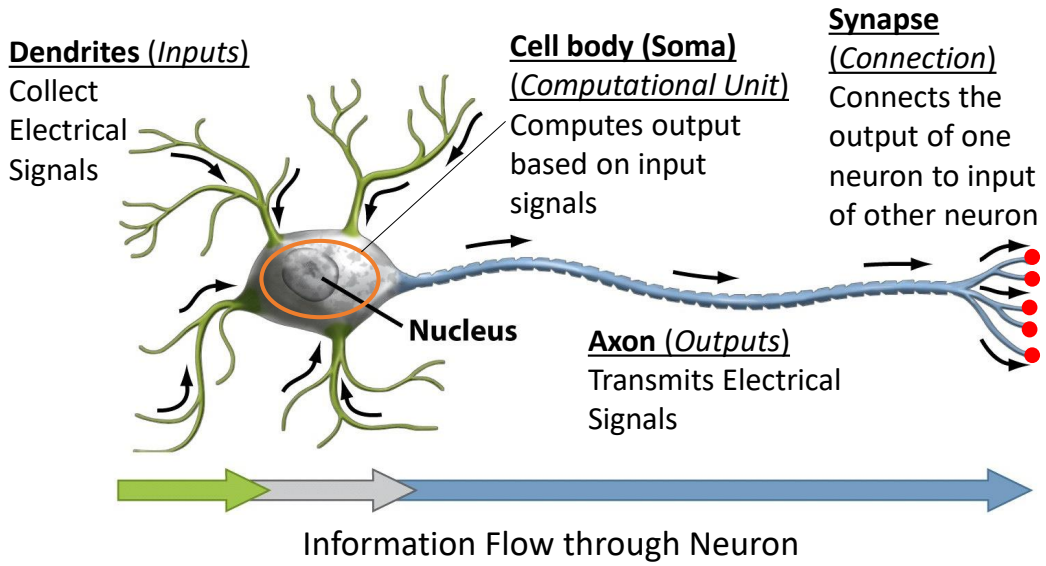


Fig. 1.2 Model of a biological neuron, adapted from [2] [1].

A biological neuron consists of the cell body (*soma*), the *dendrites* and an *axon* [2]. The dendrites and the axon are fibers; the former accept stimulus, then processed by the soma, while the axon carries the neuron output signal to the successive neurons. Neurons are electrically excitable, when the input voltage overextends a certain threshold, an action potential (a pulse signal) is generated and transmitted on the axon. Since the neuron can only have a full response or no response according to the input voltage value, its response is called all-or-none. The computational model has evolved over time to take the form presented below (Figure 1.3) [34][35]. The model consists of a non-linear function $\sigma(\cdot)$ applied to a weighted sum of all inputs (Eq. 1.1) to which an offset b is added (Eq. 1.2).

Artificial neural networks are contrived as directed graphs whose nodes represent the neurons. A network is called a *feedforward* neural network (NN) if the graph is acyclic. On the other hand, if the graph is cyclic, then the network is *recurrent*, and it exhibits a temporal dynamic behavior.

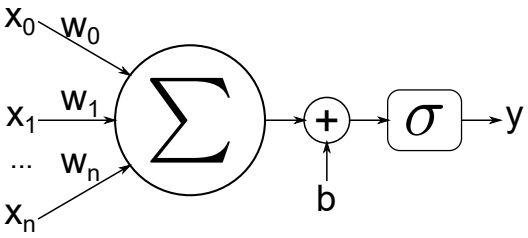


Fig. 1.3 Model of an artificial neuron. [1]

$$g(\mathbf{x}) = \sum_{n=0}^{N-1} \mathbf{x}[n]\mathbf{w}[n] \tag{1.1}$$

$$y = \sigma\left(g(\mathbf{x}) + b\right) \tag{1.2}$$

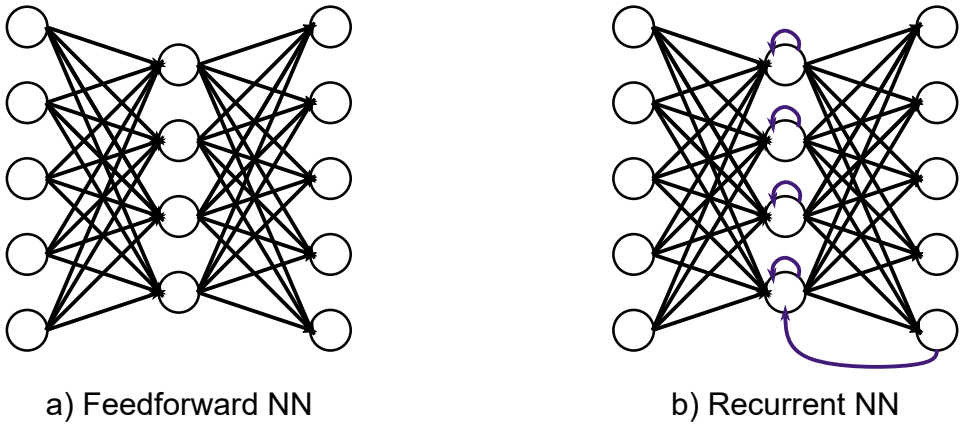


Fig. 1.4 Examples of **(a)** a feedforward NN and **(b)** a recurrent NN. [1]

As shown in Figure 1.4, the NNs are organized in layers of nodes. Each layer l receives input from the previous one $l-1$ and sends the output to the following one $l+1$. The first and the last layers are called *input* and *output*, respectively, while all those between them are called *hidden* layers. The number of the hidden layer determines the *depth* of a NN; with more than three hidden layers, a network is called *Deep Neural Network* (DNN) [36]. The ability of an NN to solve different types of problems lies in the ability to find optimal values for the weights and the biases of its neurons and the flexibility of the connections and structure that the various layers can assume, as explained in the next section.

1.2.1 Layers

Fully Connected (FC) layers. In a Fully Connected layer l, each neuron performs a weighted sum of all the input activations coming from the former layer $l-1$, as a result, each neuron is connected to all the previous ones:

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

$$0 \leq c_o < C_o, \;\; 0 \leq c_i < C_i$$

where C_i and C_o represent the number of neurons of layers $l-1$ and l respectively. An FC layer can be easily described using pseudocode in Fig. 1.5, considering that N is the *batch size*, or in other words the input groups that can be handled in parallel.

```
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]
```

Fig. 1.5 Pseudocode of an FC layer. [1]

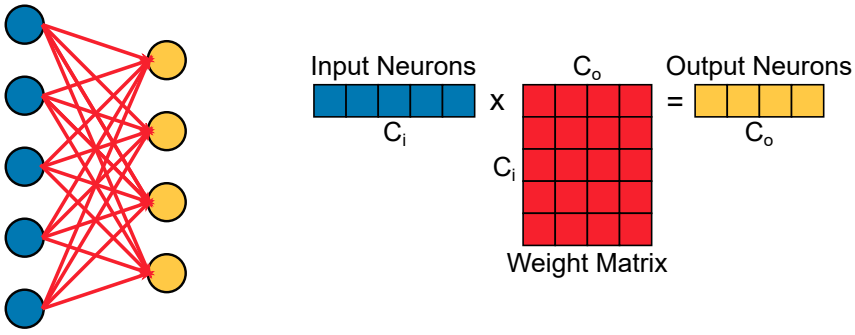


Fig. 1.6 Example of an FC layer (left) and of how it can be modeled by a vector-matrix multiplication (right). [1]

Looking at the pseudocode, it is possible to understand that the FC layer is nothing but a matrix-vector product where weights are organized according to a matrix of size $C_i \times C_o$ (see Figure 1.6).

Since this matrix can assume considerable dimensions, the number of parameters (weights) related to an FC layer can grow out of proportion. However, it is not always necessary to process all the information from the previous layer, but in many cases, a portion of it is enough; this is the reason for using convolutional layers.

Convolutional (Conv) layers. Due to the complexity and number of parameters required, FC layers are not suitable for many applications, including object detection

and recognition. Furthermore, their dense connectivity would play off in high-resolution images making the model unmanageable, not to mention that in FCs there is no distinction between elements spatially close or distant, but they are all equally treated. To overcome these problems, a new architecture, known as the Convolutional Neural Network (CNN), was proposed in 1998 [37], which includes Conv layers based on the ideas of *local receptive fields* and *shared weights*. David H. Hubel and Torsten Wiesel [38] proved the concept of local reception fields by studying the visual cortex of a cat. They showed that different groups of neurons were stimulated depending on whether the cat was visually exposed to images with vertical lines or horizontal lines. As a result, it is inferred that *locally sensitive* neurons are exposed to small portions of the visual field, while other higher-level neurons handle more significant image portions to dissect more complex patterns. By adopting that concept to neural networks, the neurons are reorganized into a 2D structure, called a *feature map*, in which every neuron in a layer l is no longer directly connected to the entire previous layer $l-1$, but only to a sub-portion of dimension $[H_k \times W_k]$. The size of the receptive field (weight matrix) is generally called *kernel size*, while the distance between two adjacent receptive fields is defined as *stride S*. Thanks to the shared weights, all neurons belonging to layer l have the same weight matrix, extracting a particular feature from layer $l-1$. In order to detect multiple features, a Conv layer is composed of several feature maps called channels and several kernels.

A Conv layer performs a convolution between a *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 $[C_o]$ producing as a result an *output feature map* **Ofm** of size $[C_o \times H_o \times W_o]$:

$$\mathbf{Ofm}[c_o, h_o, w_o] = \sum_{c_i=0}^{C_i-1} \sum_{h_k=0}^{H_k-1} \sum_{w_k=0}^{W_k-1} \Big(\mathbf{W}[c_i, c_o, h_k, w_k] \mathbf{Ifm}[c_i, Sh_o + h_k, Sw_o + w_k] + \mathbf{b}[c_o] \Big)$$

$$0 \leq c_o < C_o, \;\; 0 \leq h_o < H_o, \;\; 0 \leq w_o < W_o$$
$$0 \leq h_k < H_k, \;\; 0 \leq w_k < W_k$$

Figure 1.7 shows the pseudocode of a Conv layer, and Figure 1.8 gives a graphical representation.

```
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]
```

Fig. 1.7 Pseudocode of a Conv layer. [1]

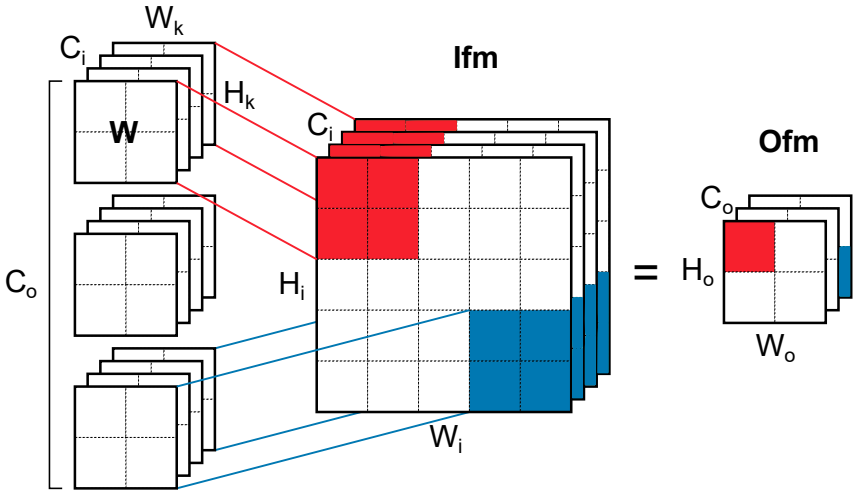


Fig. 1.8 Graphical representation of the convolution operation in a Conv layer. A sub-tensor of **Ifm** (red) is multiplied by a sub-tensor of **W** and the results are accumulated to produce a single value (red) of the **Ofm**. [1]

Pooling layers. Generally, Pooling layers come after Conv layers downsampling feature maps by summarizing statistics in their receptive fields. For the group of neurons in each receptive field, they yield a single value featuring a specific statistic of the group such as the average or the maximum value, Figure 1.9. In the Pooling case, the stride is usually equal to the size of the receptive field, providing non-overlapping windows.

Pooling layers downsize the dimension of the feature map reducing the number of activations. This represents a benefit for the memory requirements and the number of operations to be performed. Moreover, while Conv layers are susceptible to spatial variations on the input, the downsampling process allows pooling layers to make the output more robust, invariant to small local translations.

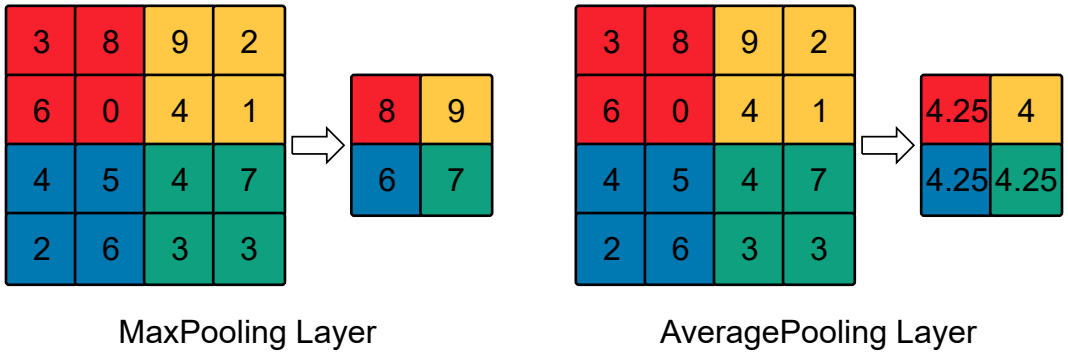


Fig. 1.9 Examples of MaxPooling (left) and AveragePooling (right) layers. [1]

Normalization layers. DNNs are long sequences of layers, so it is crucial to have some numerical stability as processing progresses into deeper layers. Usually, neural network inputs are preprocessed to present a normal distribution, namely zero mean and unit variance. Normalization is essential to hold different inputs in the same ranges in order to be analyzed by the same NN model. In addition, as will be described later, some layers use non-linear functions, so having zero-centered values allow avoiding early saturation of activations. In order to apply the same normalization constraint to internal activations, normalization layers are inserted between the Conv and FC layers. It should also be noted that normalized activations speed up training, as the layers do not need to adjust to different distributions at each training step.

The commonly adopted normalization method is Batch Normalization (BatchNorm) [39](Eq. 1.3). The operation performed by the BatchNorm layer is standardization:

$$y = \frac{x - E[x]}{Var[x] + \varepsilon} \cdot \gamma + \beta \tag{1.3}$$

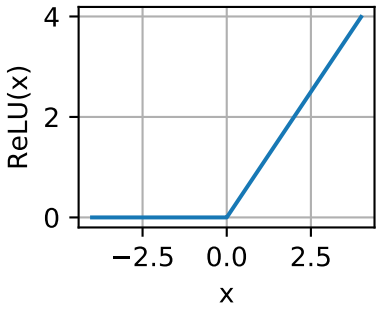
Where $E[x]$ and $Var[x]$ represent the mean and standard deviation of the input tensor x, respectively. While ε is a value necessary for numerical stability, γ and β

are two trainable parameters of the BatchNorm that are tuned during the training process.

Non-linear activation functions. Without non-linear activation functions, the NN would be a simple chain of linear algebra operations, unfit to break complex non-linear problems. Thus, 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 activation to be greater than or equal to zero. It is very common as it is computationally efficient since it requires a simple comparison between x and 0.

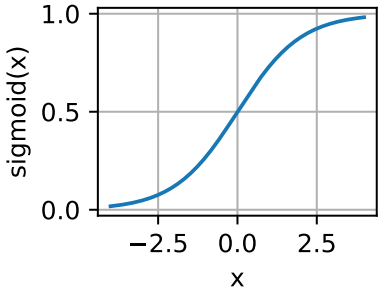
$$y = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$



  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 $x < 0$, while the latter exploits a log curve when $x < 0$. These alternatives have been introduced to sort out the *dying ReLU* problem, in fact, since the slope of the ReLU for $x < 0$ is zero, the neurons in this region are not trained. In addition, Leaky-Relu and ELU are more balanced towards zero, speeding up the training.

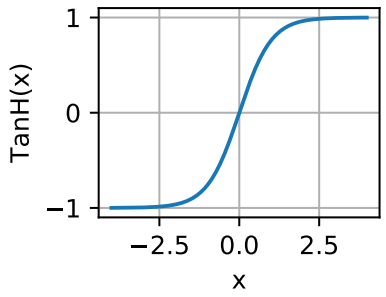
- *Sigmoid* function normalizes the output in the range $(0, 1)$. Contrarily to the ReLU function, it is computationally expensive:

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



- *Hyperbolic Tangent* function (TanH) normalizes the output 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 entries as input: each entry is normalized in the range $(0,1)$, and the total sum of all vector elements is equal to 1. This function is mainly used as an output layer if the output represents the classification probabilities.

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

1.2.2 Training vs Inference

A NN can learn to unravel a problem by tuning the values of the weights and biases of its layers, this is known as *training*. On the other hand, it is possible to use a trained NN with pre-learned weights and biases to obtain a prediction. This is known as *inference*. There are different ways of training a NN (see Figure 1.10):

Supervised learning: It requires a collection of labeled input-output pairs, i.e., a set of inputs (*data*) with the corresponding expected output (*labels*). This collection of pairs is called a *training set*. The model receives a labeled input during the supervised learning and tunes its parameters based on the difference between the expected output and the actual output (error). Supervised learning is primarily employed today in an extensive range of applications, thanks to datasets' immense availability and significant performances in the big-data era.

Unsupervised learning: It is performed when merely non-labeled data are available. It looks for common patterns in the data. An illustration of unsu-

pervised learning is *clustering*, which clusters data based on their common attributes. Neural networks that apply unsupervised learning are, for example, *autoencoders* and *Generative Adversarial Networks* (GANs).

Reinforcement learning: Reinforcement learning, as well as unsupervised learning, does not need labeled data. This type of learning aims to create an autonomous agent able to make decisions in a given environment. The training scenario is composed of the agent who takes action in an environment, then an interpreter evaluates the agent's actions in terms of a reward. The evaluations are then fed back to the agent. The agent tries to maximize the reward.

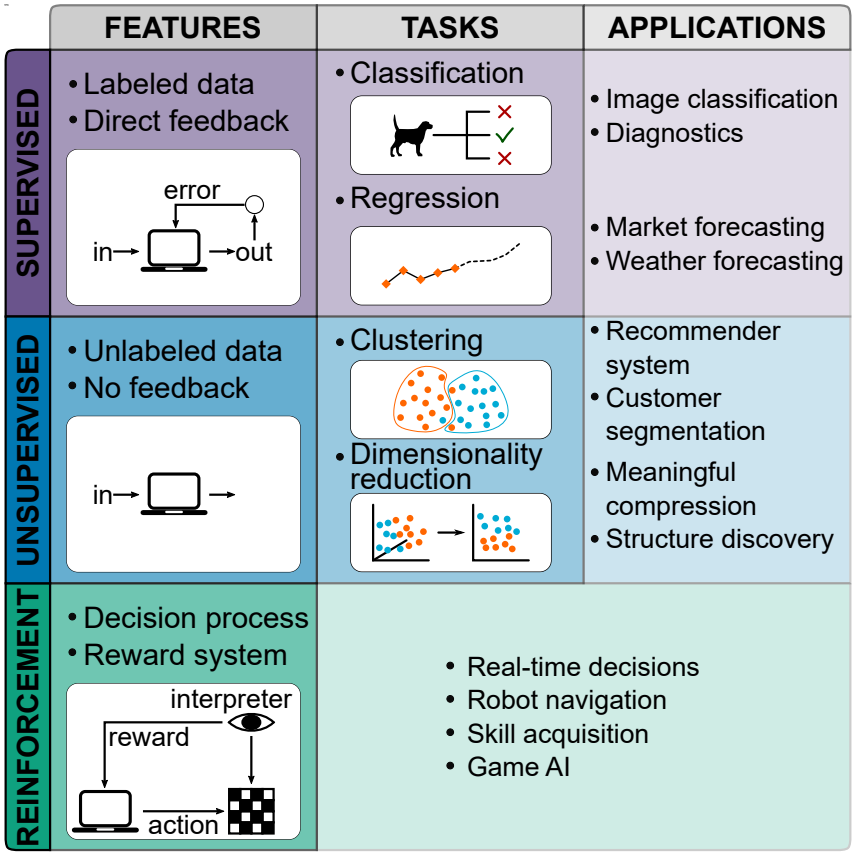


Fig. 1.10 Features, achievable tasks and applications of the three existing ways of training (supervised, unsupervised, and reinforcement) [1]

Gradient backpropagation is the most commonly used supervised-learning algorithm for training DNNs. The input sample is fed into the DNN that performs an inference. The output prediction is then compared to the expected output, and a *loss function*, such as Euclidean distance or Mean Squared Error (MSE), calculates the

loss (L). The learning process takes place by updating the DNN weights of a quantity proportional to the partial derivative of the loss concerning weights themselves, namely the gradient. By applying the *chain rule* of calculus o calculate the derivative of the loss starting from the output of the network, it is possible to backpropagate the gradients from the output to the inner layers (backpropagation algorithm).

The learning consists of updating the weights and biases of the network; this process can be done with different optimization algorithms. The most straightforward one is gradient descent (GD), shown in Eq. 1.4, 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 [40], Nesterov accelerated gradient [41], Adagrad [42], Adadelta [43] and Adam [44].

$$\theta^{t+1} = \theta^t - \eta \frac{\partial L}{\partial \theta^t} \tag{1.4}$$

During the training, a neural network could run into the overfitting problem, i.e., a complex model with many parameters could fit the data of the training set too accurately. Therefore, the model "memorizes" the correct outcome for each input rather than learning to generalize; this translates into poor performance on datasets that have never been seen before. To avoid the overfitting problem, the number of parameters can be reduced, having a simpler model, or a *regularization technique* can be employed. L1 and L2 [45] are standard regularization techniques, both consist of adding a regularization term to the loss function, which has the consequence of reducing the magnitude of the weights. This results in a compressed and simpler model. Another good technique is *dropout* [46], i.e., at each iteration, during the inference, some neurons are randomly selected and switched off (removed) from the model.

1.2.3 DNN Models

In the last decades, numerous CNNs models have been proposed with the aim to meet the best-possible performance for a given task. Figure 1.11 shows a timeline of influential neural network models with their classification accuracy for image classification task on the ImageNet dataset and number of parameters. These models are examined in the following paragraphs and compared in Table 1.1.

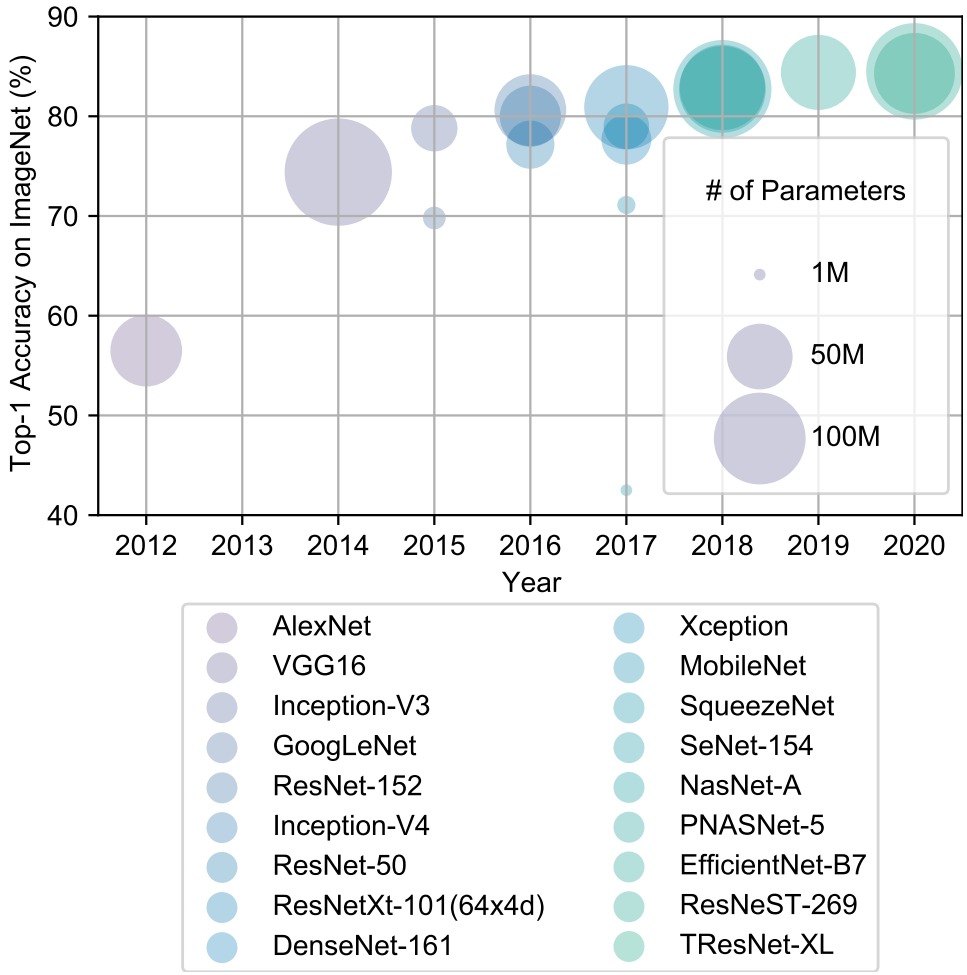


Fig. 1.11 Timeline of significant neural networks models with the accuracies achieved on the ImageNet dataset and the number of parameters. [1]

LeNet [37] (1998): One of the first NN to introduce a convolutional structure has inspired the following research on CNNs, and one of the first models to be trained with backpropagation. It was conceived to recognize handwritten digits characterized by 32×32 pixels images. One of the most known versions is LeNet-5, which consists of five layers: while the first two are convolutional layers, the remaining three are FC layers. The Conv layers are characterized by 5×5 kernels and are followed by 2×2 average pooling layers. The dominant activation function is the hyperbolic tangent (tanh), except for the output layer that applies the softmax function.

AlexNet [47] (2012): It is the first CNN to win the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC-2012) [48], achieving consistent accuracy advancements compared to the non-convolutional networks winners of the former

editions. Its design is conceived to be trained on 256×256 pixels high-resolution images, part of the ImageNet dataset, a database of more than 1.3M images divided into 1000 classes. Following the same principle of LeNet, AlexNet stacks 5 consecutive Conv layers and three FC layers. Moreover, AlexNet introduces the Rectified Linear Units (ReLU) as the activation function for the first time, significantly reducing the training phase.

Table 1.1 Comparison of the models presented in Section 1.2.2

Model	**Year**	**Contribution**
LeNet [37]	1998	-First popular CNN
AlexNet [47]	2012	-First CNN winner of ILSVRC -Introduction of ReLU
VGG16 [49]	2014	-Smaller kernels
GoogLeNet [3]	2015	-Inception block
ResNet [4]	2016	-Skip connections -Residual learning
ResNetXt-101_64x4d [50]	2017	-Grouped convolution
DenseNet161 [5]	2017	-Regular structure
CapsNet [7]	2017	- Capsules - Dynamic Routing
SeNet154 [6]	2018	-Dependencies between feature maps are exploted
NasNet-A [51]	2018	- Neural Architecture Search - Transfer learning

VGG [49] (2014): The trend of model development initially followed the development of hardware, proposing increasingly expensive models from the computational point of view. VGG model following the structure of AlexNet furtherly increases the number of Conv layers. In particular, VGG-16 has 13 Conv layers and three FC layers, while VGG-19, with 19 layers, was the winner of ILSVRC-2014.

GoogLeNet [3] (2015): This model is based on the inception module, a dense structure, as a building block for layers. An inception module (see Figure 1.12) extracts features at various scales and concatenates them at the output, forwarding them to the next layer. As deeper and deeper models were built, the accuracy gradually increased, however, beyond a certain depth, the problem of *vanishing gradient* becomes considerable. During backpropagation, the gradients are computed

exploiting the chain rule, assuming values in the range $[0,1]$ or $[-1,1]$; thus, the magnitude of the gradients decreases exponentially with the depth of the network. In the first layers, gradients can become small enough to prevent proper training. To prevent this problem, GoogLeNet added two classifiers used for training only that take the activations at network earlier stages to increase the magnitude of their gradients. GoogLeNet successors are Inception-v3 [52] and Inception-v4 [53].

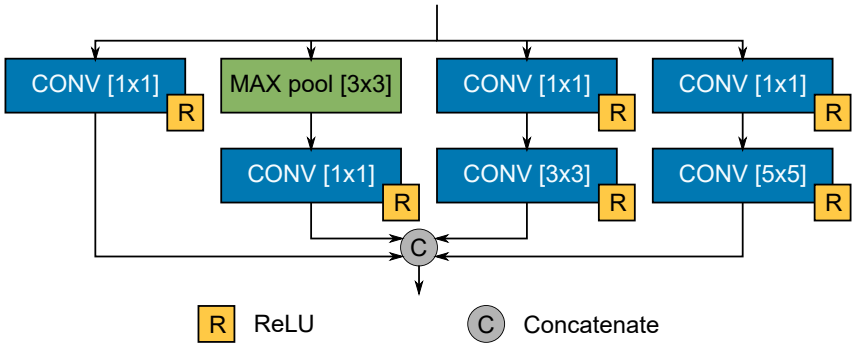


Fig. 1.12 Inception Module used in GoogLeNet [3]. Convolutions are performed in four parallel branches and the four outputs are concatenated to produce the single output of the Inception module. [1]

ResNet [4] (2015). Residual Networks (ResNets) have popularized the *skip connections*, a connection that runs in parallel to the Conv layer that mitigates the problem of vanishing gradients. Moreover, ResNets are the first architectures to employ batch normalization layers. Different ResNet-based models with higher accuracies have been proposed over the years, such as ResNetXt [50], ResNeST [54], or TResNet [55].

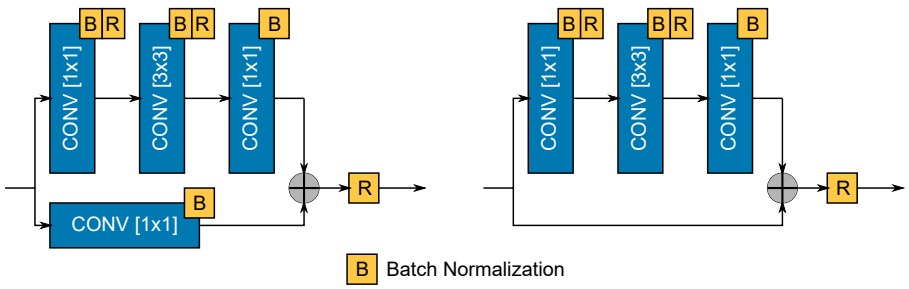


Fig. 1.13 Skip connection modules used in Residual Networks [4]. Three convolution are performed in series and a parallel connection is added. In the parallel connection, it is possible to choose between a 1×1 convolution (**left**) or the identity function, i.e., no operation (**right**). The results of the two branches are summed. [1]

DenseNet [5] (2016). Exploiting the success of skip connections, DenseNets adopt a regular and therefore more straightforward connection pattern based on Dense Block, as shown in Figure 1.14. In a Dense Block, each layer input is a concatenation of the activations of all the preceding layers. A DenseNet comprises Dense Blocks of different depths, interleaved by Conv and Pooling layers for dimensionality reduction.

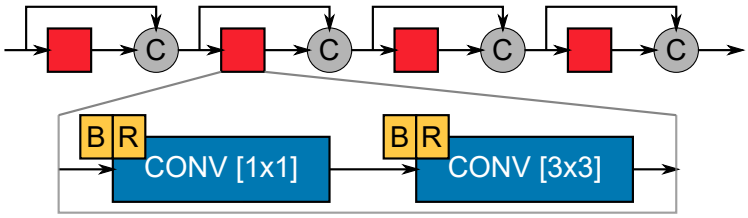


Fig. 1.14 Dense blocks used in DenseNets [5]. The output and input of each blue block, that is the series of two convolutions, are concatenated. [1]

SENet [6] (2017). Squeeze-and-Excitation Networks (SENets) model the relationship between the different channels present in the feature maps reworking classic layers, e.g., Conv layers or blocks, e.g., inception or residual modules. Figure 1.15 shows how the SE approach alters a residual module. SENet-154 is the NN winner of ILSVRC-2017, which is built integrating SE blocks in a version of ResNetXt [50].

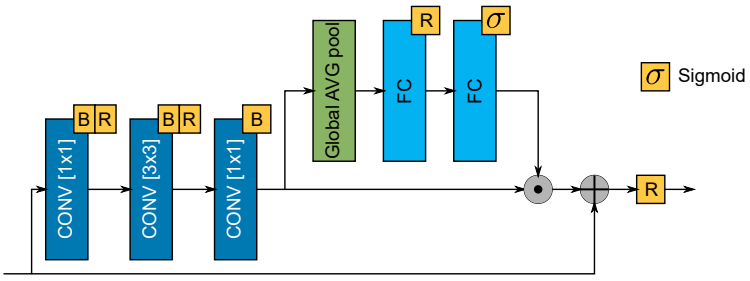


Fig. 1.15 Residual module as modified in Squeeze-and-Excitation Networks [6]. A skip connection is inserted in parallel to a pooling and two FC layers, and the output of the two branches are multiplied. As in traditional residual modules, a skip connection runs in parallel to the whole block. [1]

Capsule Network (2017). The Capsule Networks were created to unravel some of the issues of CNNs, such as the loss of data caused by the downsampling present in pooling layers or the elevated sensitivity to input rotations or shifts. The main idea of *capsules* was presented in [56], while the first network model was proposed in [7]. CapsNet replaces the neurons with capsules, i.e., a vector of neurons. Each

entry of the input vector encodes an instantiation parameter, such as the rotation or the width of a certain entity, while the number of entries in the vector represents the instantiation probability of the same entity. As described above, the length of the vector represents a probability; thus, its value must be in the range $[0,1]$. Based on this reason, capsule layers exploit the *squash* function (Eq. 1.5) as non-linear activation function.

$$\vec{y} = \frac{|\vec{x}|^2}{1+|\vec{x}|^2}\frac{\vec{x}}{|\vec{x}|} \tag{1.5}$$

Furthermore, Capsule Networks substitute the pooling layers with a *dynamic routing algorithm* that reinforces the links between capsules of adjacent layers if relevant entities are detected. Figure 1.16 shows the Capsule Network model as proposed in [7]. The work in [57] proposes to arrange the values of the capsules in matrices, while the dynamic routing is substituted by the *EM routing*.

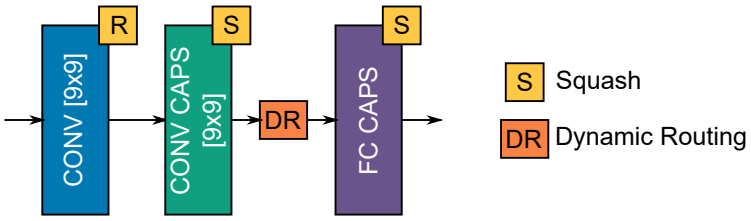


Fig. 1.16 Capsule Network [7]. [1]

NASNet [51] (2018). NASNet is the first popular NN model designed with neural architecture search. NasNet was created by searching a simple cell for a dataset in a small search space. The complexity of the model can be determined by the number of cells stacked together. Other models resulting from neural architecture search are PNASNet-5 [58] and EfficientNet [59].

1.3 Contributions and Organization

This thesis work mainly proposes three contributions ranging from the software and modeling side of Deep Learning to the implementation and verification of two hardware accelerator architectures. The objectives of this dissertation are as follows:

- To adapt an alternative to backpropagation, such as the Extreme Learning Machine (ELM), to the Continual Learning (CL) scenario in order to achieve a faster approach capable of learning progressively from a continuous data stream without the help of any regularization technique.
- To propose a low-power bit-serial accelerator for edge computing integrated into PULPissimo with a focus on power efficiency.
- Propose a variant of the previous architecture that can exploit the sparsity of activations to test whether avoiding null operations can bring benefits in terms of acceleration and power efficiency.

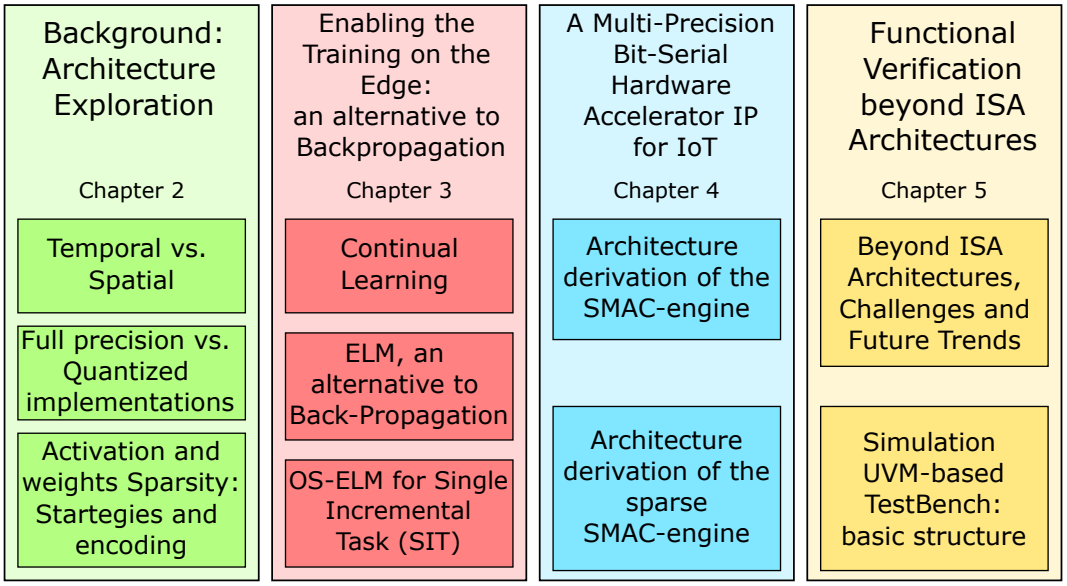


Fig. 1.17 Thesis overview

Figure 1.17 shows the organization of the thesis, which consists of four main chapters and a conclusion as described in the following.

Chapter two introduces a crucial background part related to architectural exploration for developing Deep Learning enabled hardware accelerators. In particular, it is divided into three parts. The first section explains the differences between temporal architectures (mainly CPU and GPU) and spatial architectures. This part is essential to understand how accelerator dataflows are determined, leading to subsequent architectural choices.

The second section describes the use of quantized models. Quantization is a fundamental process when deploying NNs at the edge to reduce the computational complexity and memory required. In fact, quantization allows network parameters, such as activations and weights, to be represented as integer values over a smaller number of bits. In the current state of the art, the bitwidth of activations and weights ranges from 16/8 bits to more extreme representations on 6 or 4 bits.

The third and final section discusses the encoding of activations and weights in the case of sparse networks and how sparsity can be used to improve the performance of an accelerator in terms of speed and power efficiency.

Chapter three presents an alternative to backpropagation that can be used in the case of Continual Learning to create a methodology that allows a DL model to learn continuously. The first section introduces Continual Learning by explaining what it consists of and what its challenges are: catastrophic forgetting, application scenarios, and benchmarking. It also explains what scenarios can occur in CL based on how new samples are delivered to the network that must learn them and how benchmarking is performed for these scenarios. On the other hand, the second section explains what the Extreme Learning Machine consists of, promoting its pros and cons, and a variant of it: the Online Sequential ELM.

In the third and final section, the performance of the OS-ELM is carefully evaluated in terms of processing time, batch size, and sorting resulting in outstanding flexibility and speed. This algorithm appears to be up to 19x faster than backpropagation when employed on a commercial CPU. Moreover, it can work with batch sizes ranging from the entire dataset size to batches of one sample.

Finally, the OS-ELM is adapted and employed in a CL scenario in which each batch contains a new class. The results show that such an algorithm is comparable with state-of-the-art, even managing to overcome it.

Chapter four consists of 2 macro sections. The first describes developing and implementing a hardware architecture for CNN acceleration focused on the area, and power optimization called Serial Multiply and Accumulate engine (SMAC engine). The purpose is to develop a low-power architecture for running CNN on edge. The accelerator follows a bit-serial approach, which means that multiplications between activations and weights are performed bitwise. This approach allows heavy area reduction by replacing multipliers with simple AND gates at the expense of latency. In fact, performing multiplications serially requires a higher number of clock cycles

to complete the operation. In addition, the parallelism of the architecture is designed to be flexible by accommodating 8/4 bit activations and 8/6/4 bit weights. The serial approach, as opposed to the canonical approach, allows the data bitwidth to vary while still maintaining 100 percent hardware utilization. The 'accelerator interfaces are designed so that they can be integrated into PULPissimo. Results show that the SMAC engine can achieve a peak throughput of 18.75 GMAC/s and power efficiency of 0.58 pJ/MAC.

On the other hand, the second macro section describes the further evolution of the previous architecture. Here the goal is to try to further improve the performance of the accelerator by trying to leverage data sparsity. In fact, sparsity in a DNN can range from 50 to over 80 percent. All this null data does not contribute to the final result. Implementing the ability to identify and avoid operations involving null elements could improve performance in terms of execution latency and consequently save energy. However, this ability requires an overhead of area and power dictated by the data control logic. This section describes the development and evolution of the SMAC engine to accommodate the ability to exploit the sparsity of activations, derived more from the ReLU activation function, to further speed up execution. The results are presented incrementally from the previous implementation.

Chapter five briefly introduces the challenges and new trends in the verification world. This part of the dissertation aims to illustrate the importance of verification and explain how verification has been applied for the architectures presented in Chapter four.

In non-ISA-based architectures, verification is becoming highly complex and requires more effort than design (about 60% versus 40%). This chapter describes how we can reduce verification time, to enhance the tool scalability and security issues. In addition, a verification trends section attempts to predict what will happen in the next few years in this area.

Finally, what a UVM-based Testbench simulation consists of is described. After a detailed description of each component, a brief description of how this testbench is used is presented.

Chapter six concludes the dissertation, summarizing the main achievements and contribution of our research.

Table 1.2 gives a list of all the acronyms used in the dissertation.

Table 1.2 List of Acronyms.

A	Activation	LIM	Logic-in-Memory
AI	Artificial Intelligence	MAC	Multiply-and-Accumulate
ALU	Arithmetic Logic Unit	ML	Machine Learning
ASIC	Application Specific Integrated Circuit	MSE	Mean Squared Error
BC	Binary Connect	NN	Neural Network
BLAS	Basic Linear Algebra Subroutines	NoC	Network-on-Chip
BN	Batch Normalization	NPU	Neural Processing Unit
BNN	Binarized Neural Network	NLR	No Local Reuse
BWN	Binary Weight Net	OFM	Output Feature Map
CIS	Compressed Image Size	OS	Output Stationary
CL	Continual Learning	PE	Processing Element
CNN	Convolutional Neural Network	QWN	Quatized Neural Network
Conv	Convolutional	ReLU	Rectified Linear Unit
CP	Canonical Polyadic	RF	Register File
CPU	Central Processing Unit	RLC	Run Length Coding
CSC	Compressed Sparse Column	RS	Row Stationary
CSR	Compressed Sparse Row	RTL	Register Transfer Level
CWR	Copy Weight with Reinit	SIMD	Single-Instruction Multiple-Data
DL	Deep Learning	SIMT	Single-Instruction Multiple-Threads
DNN	Deep Neural Network	SIT	Single Incremental Task
DRAM	Dynamic Random Access Memory	SNN	Spiking Neural Network
DUT	Device Under Test	SoC	System on a Chip
ELM	Extreme Learning Machine	SRAM	Static Random Access Memory
FC	Fully Connected	STDP	Spike Time Dependent Plasticity
FIFO	First In First Out	TB	TestBench
FM	Feature Map	TCDM	Tightly Coupled Memory
FPGA	Field Programmable Gate Array	TPU	Tensor Processing Unit
FSM	Finite State Machine	TTFS	Time To First Spike
GAN	Generative Adversarial Network	TWN	Ternary Weight Net
GD	Gradient Descent	UVM	Universal Verification Methodology
GLB	Global Buffer	VLSI	Very Large Scale Integration
GeMM	General Matrix Multiplication	W	Weight
GPU	Graphic Processing Unit	WS	Weight Stationary
HDL	Hardware Description Language		
IFM	Input Feature Map		
IP	Intellettual Property		
IoT	Internet-of-Things		
ISA	Instruction Set Architecture		
L	Loss		
LIF	Leaky-Integrate-and-Fire		

Chapter 2

Background: Architecture Exploration

2.1 Introduction

This chapter introduces a crucial background part related to architectural exploration for Deep Learning acceleration. In particular, it is divided into three parts. The first section explains the differences between temporal architectures (mainly CPU and GPU) and spatial architectures. The second section describes the use of quantized models. The third and final section discusses the encoding of activations and weights in the case of sparse networks and how sparsity can be leveraged to improve performance. The objective of this chapter is to summarize the state-of-the-art methodologies and give the necessary background to understand the upcoming chapters.

2.2 Temporal vs Spatial architectures

In Neural Network algorithms, two different types of parallelism can be identified [60]. Since the Multiply-and-Accumulate (MAC) operations performed by the neurons have no data dependencies, they can be executed in parallel. As a result, layers, such as FC and Conv layers, have *topological parallelism*. On the other hand,

samples from a training set are not fed one at a time, but the network processes them in batches (*operational parallelism*).

In order to accelerate the execution of NNs, hardware devices can exploit the layer's topological parallelism to implement parallel computing paradigms. Among the various solutions for parallel computation, the most common are temporal and spatial architectures [61]. Both the architectures are composed of a large number of Processing Elements (PEs) that execute operations in parallel on different or the same data. In temporal architectures, the PEs are only allowed to access data from the central memory, there are no inter-PEs connections, and the control is centralized. On the other hand, in spatial architectures, each PE features its control logic and some local memory locations. Moreover, in spatial architectures, the PEs are interconnected to trade data, creating a processing array. Figure 2.1 shows the main differences between temporal and spatial architectures.

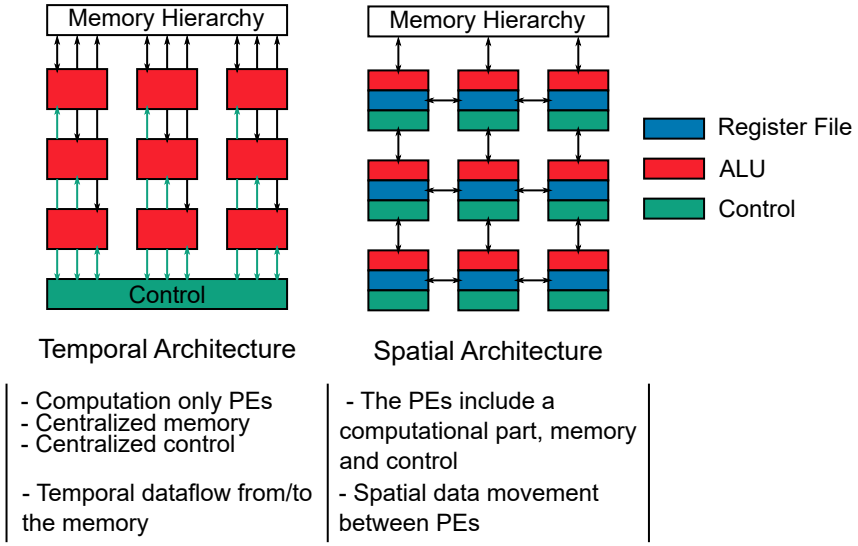


Fig. 2.1 Basic models of temporal **(left)** and spatial **(right)** architectures. [1]

2.2.1 Temporal architectures and software optimizations

Temporal architectures are typically adopted in general-purpose devices, such as CPUs and GPUs. Nowadays, many CPUs can embed *vector processors* (e.g., Intel's Xeon Phi x200 and Skylake-X CPUs) with the ability to work with multiple data segments simultaneously. Vector processors exploit multiple Arithmetic Logic Units (ALUs) that operate synchronously and perform an instruction on a data

vector. Therefore, vector processors employ the Single-Instruction-Multiple-Data (SIMD) paradigm. Among the available hardware platforms, CPUs are frequently the least used for DNNs inference or training, as they provide poor FLOPS and FLOPS/WATT performance (see Figures 2.2 and 2.3). 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 [62] 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 [63].

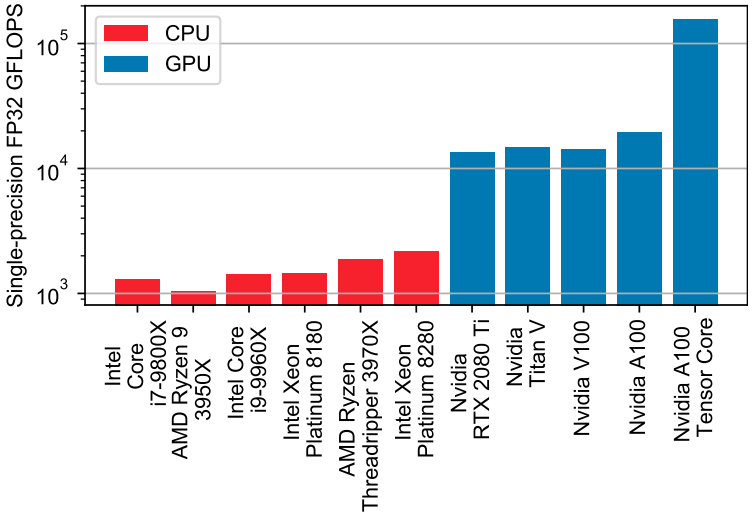


Fig. 2.2 GFLOPS comparison between different CPUs and GPUs. [1]

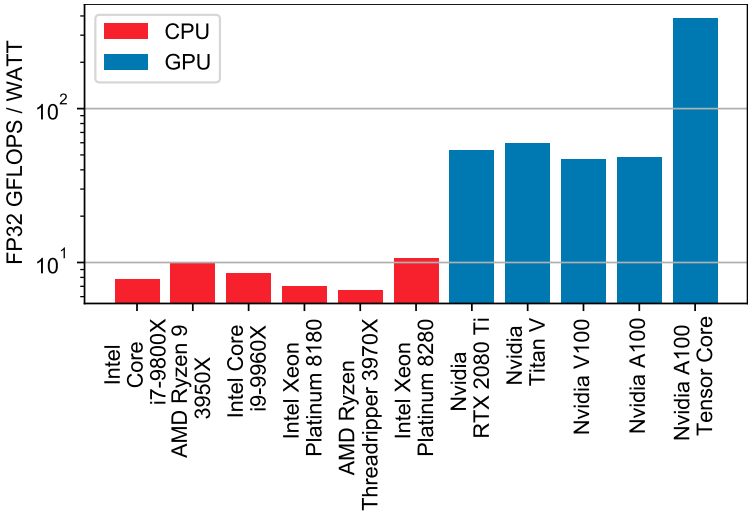


Fig. 2.3 GFLOPS/WATT comparison between different CPUs and GPUs. [1]

GPUs are manycore architectures composed of thousands of cores designed explicitly for parallel computation (e.g., 5120 cores in Nvidia V100 GPU [64]). Similar to vector CPUs, GPUs embrace the Single-Instruction-Multiple-Thread (SIMT) paradigm, introduced by Nvidia. The SIMT model executes a single instruction concurrently on multiple cores. Each core acquires different data segments that belong to multiple threads running in parallel. Nowadays, GPUs are fundamental for DNNs training and, in some instances, for inference. As one of the leading manufacturers, Nvidia has focused on GPU hardware and software codesign for DL. One of the significant advantages of Nvidia GPUs is cuDNN [65], an optimized library of DNNs primitives. cuDNN is not the only library for DL, instead, all Nvidia libraries for DNN/ML are collected in CUDA-X AI [66]. In the latest high-end GPUs, Nvidia has combined traditional CUDA Cores with Tensor Cores [64] for optimized large matrix operations and mixed-precision operations.

Concerning the software level, several libraries have been developed 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 cIBLAS). Among the considerable subroutines implemented, the BLAS includes element-wise matrix multiplication, matrix-vector multiplication, and matrix-matrix multiplication, also known as General Matrix Multiplication (GeMM).

Conv layer computation optimization is a more challenging task. Even though the operations between a weight kernel and the subsets of the input feature maps are represented as simple point-wise matrices multiplications, the memory access pattern is complex. Figure 2.4 shows how, if an input feature map is stored by rows, it is necessary to perform accesses to non-contiguous locations of memory.

In order to optimize the Conv layers, several BLAS-based computational transforms have been proposed. Numerous of the above-mentioned software libraries decline the convolution into a GeMM as proposed in [67] [68] and shown in Figure 2.5. A 4D-tensor of weights is flattened to a 2D matrix, while the data in the input feature maps are replicated and rearranged following a pattern that leads to convolution by performing matrix multiplication. This technique is extremely 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

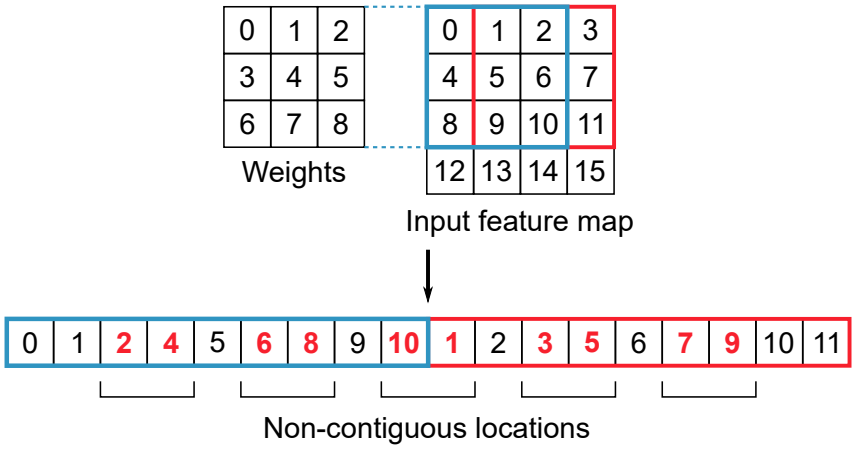


Fig. 2.4 Discontinuity in memory accesses when performing convolution on a matrix stored by rows. [1]

from $C_i \times H_i \times W_i$ to $C_i H_k W_k \times H_o W_o$. Therefore, a large memory for temporary allocation is required.

The GeMM technique for convolution can further be optimized by involving the Strassen algorithm [69][70] that lowers the number of required multiplications by partitioning the matrices. Necessary multiplications are reduced to 1/8 at each partition, however, demanding a higher number of additions.

A completely different approach consists of applying the FFT algorithm [71] to both the input feature maps and the weights to transform them from the space domain into the frequency one. The convolution operation becomes an element-wise multiplication of matrices in the frequency domain. However, the FFT algorithm presents a high computational overhead, and its efficiency has only been proven valid for large weight kernels and unitary strides. Another widely used technique is the Winograd algorithm [72][73], which, unlike the FFT algorithm, is remarkably efficient for small kernels.

2.2.2 Spatial architectures and dataflow processing

Typically, spatial architectures are implemented on FPGAs and ASICs. These platforms allow for a design tailored to specific applications at the cost of minor flexibility. NNs are particularly suited for this category of hardware implementation

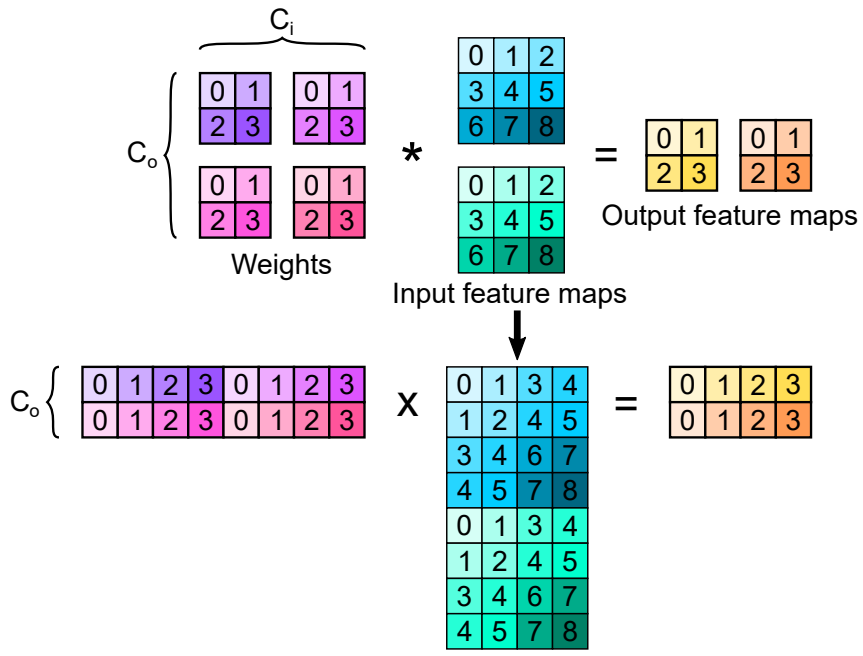


Fig. 2.5 Convolution lowering: mapping a convolution to a matrix-matrix multiplication by rearranging the matrices. [1]

since each layer's type and order of operations is stationary and known a priori. Consequently, it is possible to develop specialized and highly optimized circuits.

NNs perform simple operations, mostly multiply-and-accumulate (MACs), that nevertheless must be executed on an extensive set of data. Accordingly, the bottleneck is not the computation but the required memory accesses to read and store the inputs and results. Each MAC requires three memory read (input pixel, weight and partial sum) and a write one (updated partial sum), for a total of 4 accesses. It has been demonstrated that DRAM access has an energy cost of ~ 2 orders of magnitude higher than a MAC operation [74]. The massive DRAM access energy cost compared to the computational energy has been underlined in many state-of-the-art DNNs accelerators such as DianNao [33] or Cambricon-X [75] (Figure 2.6).

A typical DNNs accelerator hardware architecture (Figure 2.7 left) consists of:

- An **off-chip memory** (usually DRAM), storing the whole network, i.e, the weights and the activations. This represents the higher memory level, typically containing several GBs of data.

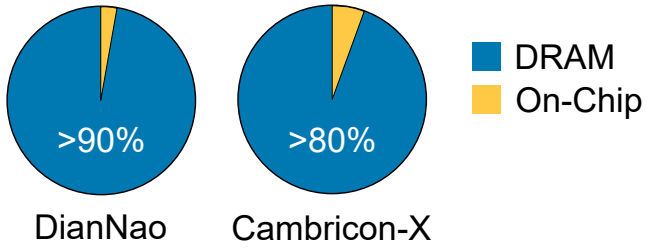


Fig. 2.6 Energy breakdown of two state-of-the-art DNNs accelerators. [1]

- An **on-chip global buffer (GLB)**, containing weights and inputs necessary to feed all the PEs. The accesses to the GLB can require two orders of magnitude less energy than that of the DRAM [76].
- An array of **PEs**, each embedding an ALU to perform MACs operations in parallel. The PEs usually incorporate one or more Register Files (RFs) to retain data locally with an energy cost-per-access lower than the GLB.
- The PEs are interconnected among them and to the GLB by a **Network-on-Chip (NoC)**. Based on how the operations are temporally scheduled and spatially mapped on the PEs, data must be transferred coordinately through the PEs to obtain the proper result. The NoC can be configured to implement manifold communication patterns, illustrated in Figure 2.7 right, depending on how data must be provided.

Given the high energy cost demanded by DRAM access, the innovation of state-of-the-art DNNs accelerators focuses on the exploitation of *data reuse*. This technique implies optimizing the architecture, mapping data on the PEs and the temporal scheduling of operations to maximize data reuse when stored in lower-level memories such as the RFs or the GLB.

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

FC layer. A FC layer consists of a matrix-vector multiplication and it, therefore, offers the possibility for *input reuse* since the input neuron vector is dot-multiplied with each row of the weights matrix of (see Figure 2.8).

Conv layer. The Conv layer has three different options for data reuse (see Figure 2.8). The weight kernel is slid over the whole input feature map to perform the convolution. *Weight reuse* is possible since the same weight kernel is multiplied

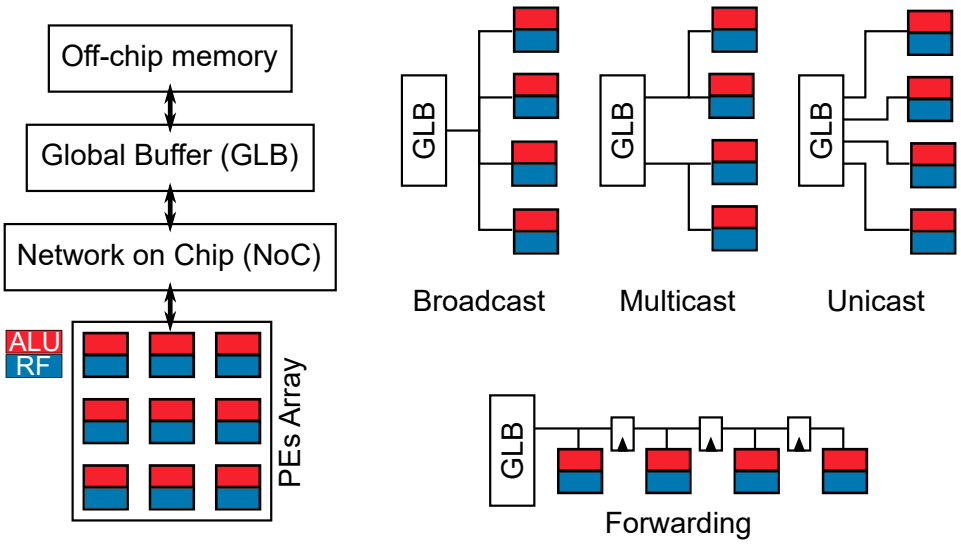


Fig. 2.7 **(left)** General structure of a hardware accelerator for DNNs and **(right)** interconnection schemes. [1]

for numerous subsets of the input feature maps. In particular, each of the C_o kernels can be reused $H_o \times W_o$ times. Since the input feature maps are employed C_o times to generate all the output feature maps, there is also an opportunity for *input reuse*. The last reuse opportunity is described as *convolutional reuse* [76]: considering the sliding window mechanism, when computing two contiguous output pixels, there is usually an intersection between the two subsets of input pixels used, as shown in Figure 2.8. The intersection's width and height depend on the kernels' dimensions ($H_k \times W_k$) and the horizontal and vertical strides (s_x, s_y). The convolutional reuse merges both the weight reuse and the input reuse.

Pooling layer. Pooling layers do not involve the use of weights. Therefore there are no opportunities for weight reuse. Furthermore, the stride parameter is usually arranged to have non-overlapping receptive fields, so no input reuse through a sliding window mechanism is possible. As a consequence, these layers do not allow for any data reuse. Given an array of PEs, each PE will perform a subset of MACs necessary to calculate the output feature map, thus a number of MACs equal to the number of PEs will be executed in parallel. The MACs have to be *spatially* and *temporally mapped* to the PEs array (Figure 2.9). The mapping defines the data movement, i.e., data load and store from/to the memory hierarchy of the accelerator and the NoC design to correctly and efficiently supply and collect inputs, weights, and partial sums. Spatial and temporal operation mapping is called *dataflow* [76].

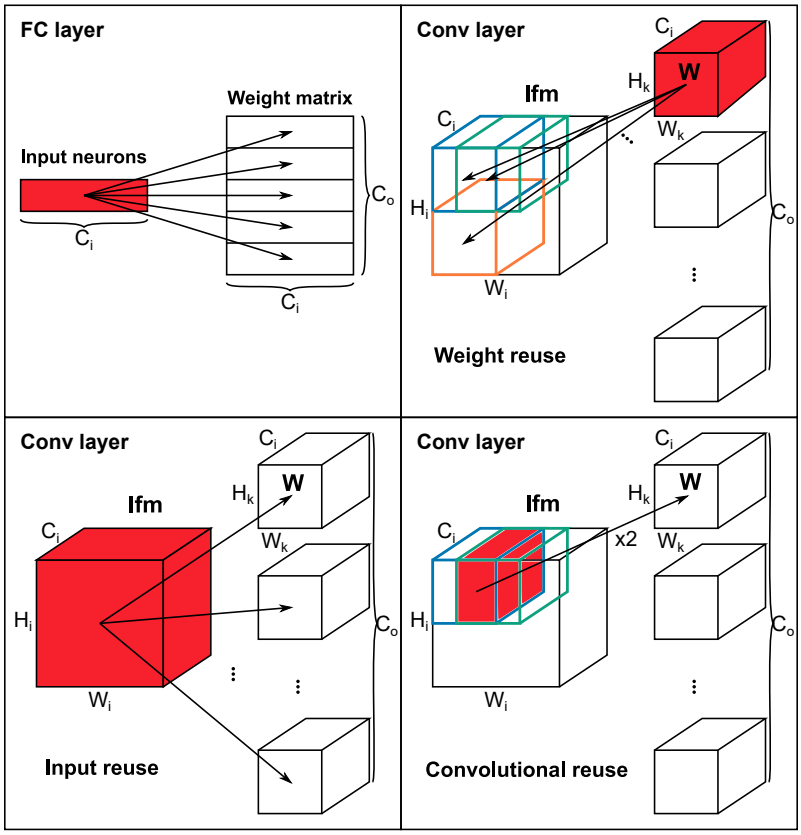


Fig. 2.8 Data reuse in an FC layer and in a Conv layer. [1]

Since the size of the PEs array and the number of MACs to be processed can be considerable, the space of possible mappings on a generic HW accelerator is enormous. Moreover, considering the energy consumption of the memory hierarchy, dataflows try to maximize the data reuse opportunities provided by the different layers of NNs to minimize off-chip memory and global buffer accesses and utilize the data stored in RFs much as possible. Chen et al. [76] introduced a taxonomy to classify existing accelerators based on their dataflow and on how they exploit data reuse, which will be explained briefly in the following.

Weight Stationary: this stationary dataflow exploits mainly the weight reuse to minimize the energy necessary to fetch the weights from the DRAM and the GLB. A subset of weights is read from the DRAM/GLB and retained locally in the RFs of the PEs. All the operations relative to a certain weight are then mapped to the PE which contains it. Figure 2.10 illustrates how operations are mapped to an array of 4 PEs to perform a 2×2 convolution on a 3×3 input feature map.

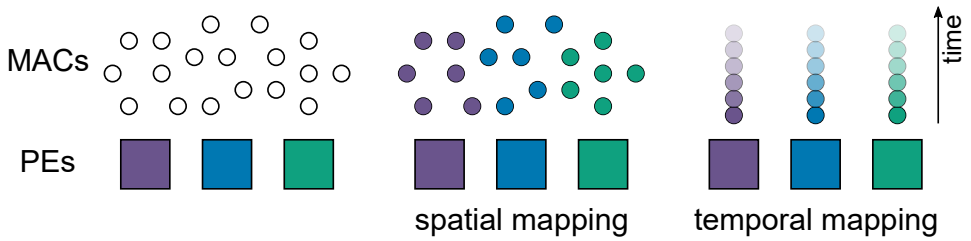


Fig. 2.9 Spatial and temporal mapping of the Multiply-and-Accumulate (MACs) operations to the Processing Elements (PEs). [1]

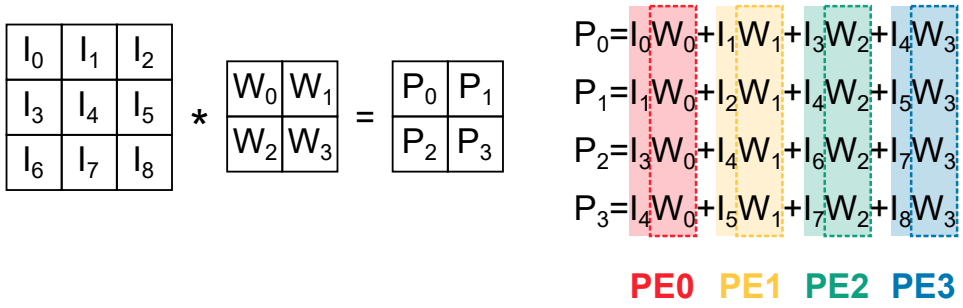


Fig. 2.10 Spatial mapping of the operations in a weight stationary dataflow. [1]

Since the weights are retained in the PEs, the inputs and partial sums movement across the array must be coordinated to optimize the utilization of the NoC. An option consists of broadcasting a single input feature map pixel to all the PEs, storing each partial sum in a register, and passing it to the neighboring PE on the right. As depicted in Figure 2.11, in certain time steps, some of the PEs perform operations that are not functional for the result (denoted in white). Moreover, at the end of each PE row, the partial sum needs to be stalled for $W_i - W_k$ cycles before being forwarded to the next one. Consequently, all of these partial sums must be stored in the GLB. The nn-X accelerator [77] features H_k FIFOs at the end of each row, each of dimension $W_i - W_k$, to introduce the appropriate delay.

The forwarding scheme can be used to move input pixels taking advantage of the convolutional and weight reuse. The forwarding scheme involves retaining the input pixel in additional local registers in the PEs and then passing it to the neighboring PEs on the right (horizontally-sliding window). Figure 2.12 depicts a dataflow with stationary weights and input forwarding. Both in [78] and [79], H_k rows of the input feature map are processed in parallel, and the partial sums of each row are then accumulated. Accordingly, the inputs are stored in H_k buffers, and the input feature

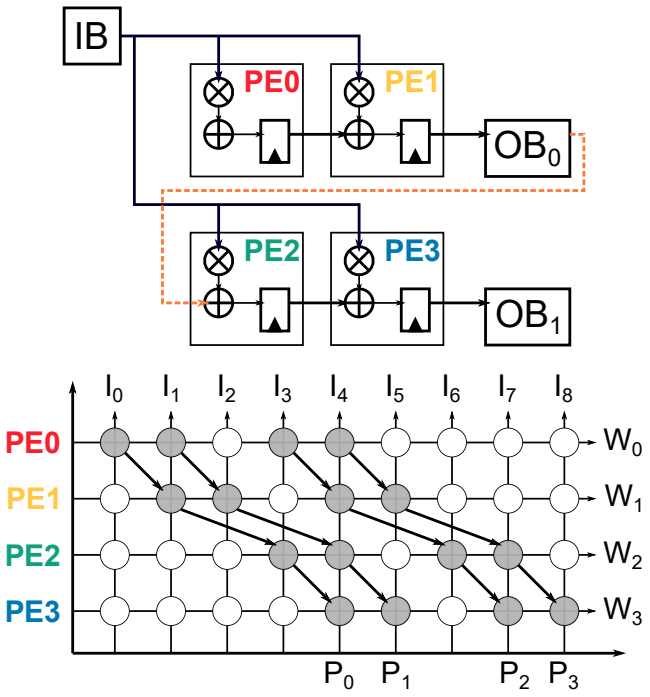


Fig. 2.11 **(top)** PEs array and **(bottom)** temporal-spatial mapping of the operations in a weight stationary accelerator with inputs broadcasting and outputs forwarding. [1]

map pixels are moved from the $(K-1)$ buffer to the 0 buffer (vertically-sliding window).

All dataflows discussed above are characterized by the fact that all operations along dimensions H_k and W_k are mapped into the 2D PE array and performed in parallel. This mapping operation is referred to as *spatial unrolling*in [80]. 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 2.13. In [80], the $H_k|W_k$ syntax is adopted to denote which loops are executed in parallel. From what concerns the software, the stationarity of the weights is instead identical to a *reorder* operation of **for** cycles, as shown in Figure 2.13. Other architectures that adopt a $H_k|W_k$ weight stationary approach are [81], [82] and [83].

By spatially unrolling the dimensions C_o and C_i ($C_o|C_i$), it is possible to obtain a different dataflow in which the weights are still stationary. As shown in Figure 2.14, the operations that must be performed are equivalent to a vector-matrix multiplication. It can be realized in hardware with a 2D systolic array. The weights are locally 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

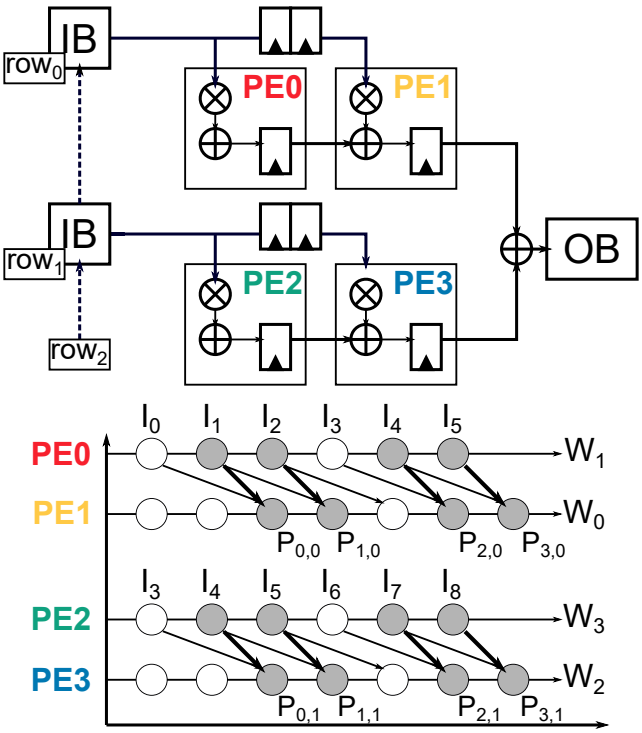


Fig. 2.12 **(top)** PEs array and **(bottom)** temporal-spatial mapping of the operations in a weight stationary accelerator with inputs and outputs forwarding. [1]

dataflow can be found in in the Tensor Processing Unit (TPU) [84] developed at Google.

Output Stationary: The output stationary dataflow focuses on minimizing the data traffic necessary to store and load the partial sums in the GLB. With the weight stationary dataflow of Figure 2.11, for example, the partial sum related to a single output pixel requires to be stored and loaded to/from the GLB $(H_k - 1) \times C_i$ times. In the output stationary dataflow, the PEs feature the possibility of locally accumulating the results of the performed MACs (Figure 2.15). Therefore, each PE produces a complete output pixel.

Even in this case, it is possible to spatially unroll the H_o and W_o loops to get an output stationary dataflow. There are several possible combinations of transmission and movement of weights and activations within the array. One of these, for example, is to transmit the activations to all PEs and then forward the weights, as shown in Figure 2.16.

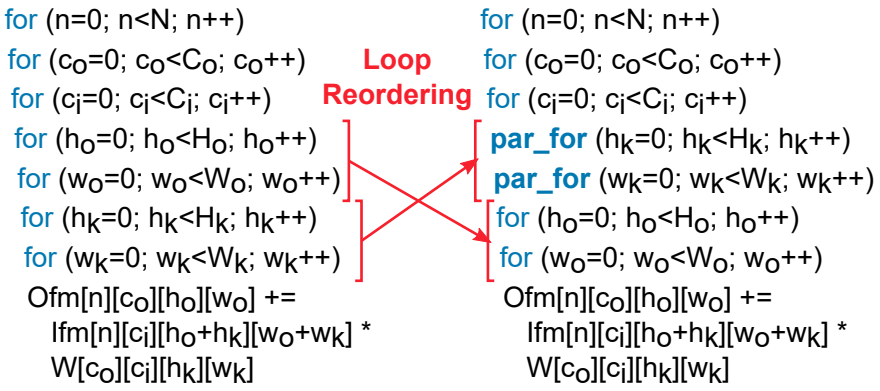


Fig. 2.13 Loop reordering of the 7-nested loops representation of the Conv layer. [1]

A famous accelerator that exploits an output stationary dataflow with $H_o|W_o$ spatial unrolling is ShiDianNao [8]. In ShiDianNao, each PE in the 2D grid processes an output pixel, and all the final results are then gathered and stored in the global buffer. A single weight is broadcasted to all the PEs at every operation cycle. Each PE can read the input activation either from the GLB, or from its right/lower neighbor. The PEs have a local RF for the partial sum accumulation and two FIFOs to store input pixels for inter-PEs communication. Computing the output pixels in parallel along the dimensions H_o and W_o is not the only possible solution for an output stationary dataflow. Origami accelerator [10], 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 [8] and [10], in [9] the output pixels along dimensions H_o and C_o are computed in parallel. Figure 2.17 graphically shows how [8], [9] and [10] spatially unroll the computation of the output pixels.

It is crucial to notice that spatially unrolling along the dimensions C_i and C_o can lead to a weight stationary or an output stationary dataflow. $C_i|C_o$ dataflow is very common because it requires a vector-matrix or matrix-matrix multiplication, and consequently, it allows to easily map both a Conv and a FC layer to the same array of PEs.

Row Stationary: The row stationary dataflow is introduced in [76] and exploited by the Eyeriss accelerator [85]. It aims to maximize the input reuse, weights, and partial sums, in contrast to the above-mentioned stationary dataflows that focus on a single type of data reuse.

All the MACs necessary to perform a row of the convolution (1D convolution) in the row stationary dataflow are mapped to a single PE. A PE has an RF to keep a row

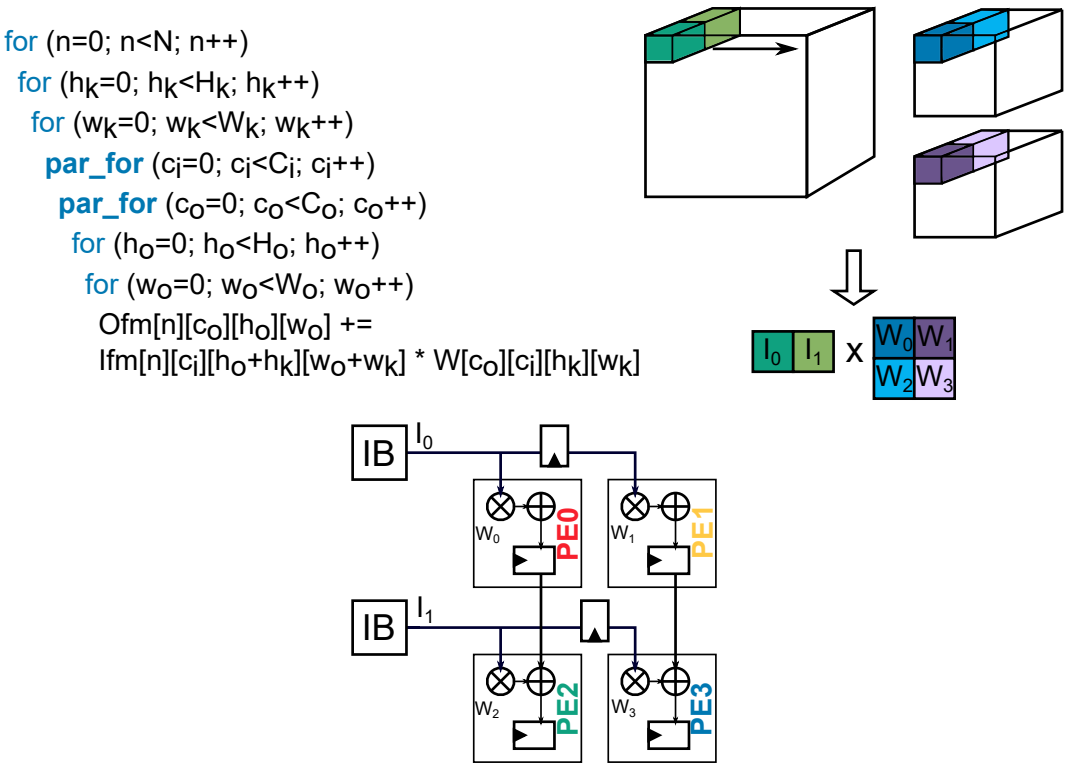


Fig. 2.14 **(top left)** Reordered loops for a dataflow with weights stationary along dimensions C_i and C_o. **(top right)** Mapping the operations of a Conv layer to a matrix multiplication. **(bottom)** PEs array in a $C_i|C_o$ weight stationary accelerator with input and output forwarding. [1]

of the weight kernel stationary while the activations are streamed in the PE using the sliding window mechanism. For 2D convolutions, a 2D array of $H_k \times H_o$ PEs is required. Each column of the array produces the $H_k \times W_o$ partial sums that contribute to a row of the output feature map. Figure 2.18 illustrates how a 2D convolution involving a 3×3 weight kernel is mapped to a row stationary dataflow.

Figure 2.18 shows 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 exact weights, the input pixels are diagonally reused, and the partial sums are vertically accumulated.

No Local Reuse. The memory elements with a low storage capacity present a higher energy efficiency, but they are inefficient in terms of area occupation (area/bit). Consequently, an RF has a higher area/bit than a scratchpad memory or an SRAM. The no local reuse dataflow minimizes the storage area by removing RFs from the

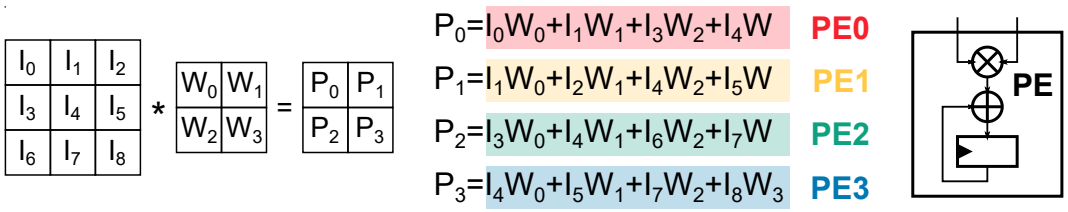


Fig. 2.15 **(left)** Spatial mapping of the operations in an output stationary dataflow. **(right)** Modified PE for an output stationary accelerator. [1]

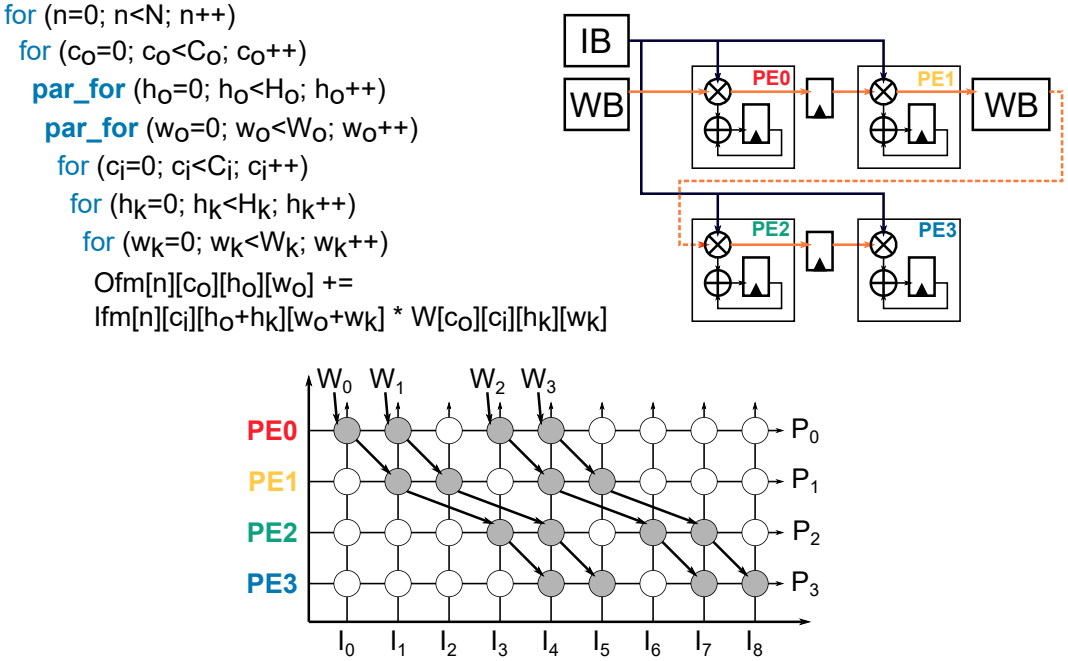


Fig. 2.16 **(top left)** Reordered loops for an output stationary dataflow . **(top right)** PEs array and **(bottom)** temporal-spatial mapping of the operations in an output stationary accelerator with input broadcasting and weight forwarding. [1]

PEs and allocating all the on-chip memory in the GLB. Without local reuse, the traffic from and to the GLB on the NoC will increase.

In the no local reuse dataflow, which dimension is spatially enrolled is irrelevant. Examples of accelerators that adopt this dataflow are [86], [33] and [87], which loop along the dimensions C_i and C_o in parallel.

A critical aspect of the dataflow definition and accelerator design is that the GLB size is usually insufficient to fully contain the input feature maps, kernel weights, and output feature maps. Therefore, the *loop tiling* technique, which consists of partitioning the larger tensors into smaller ones that can be retained in the

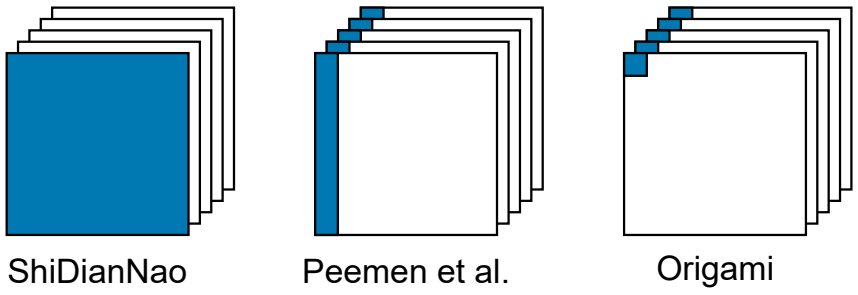


Fig. 2.17 Different solutions [8][9][10] to spatially unroll the computations in an output stationary dataflow. [1]

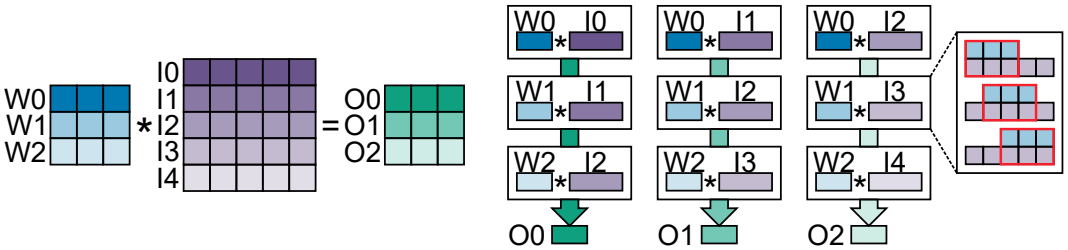


Fig. 2.18 Operations mapping in a row stationary dataflow. [1]

buffer, is necessary. The 7-nested **for** loops representing the convolutional layer are consequently split into multiple loops, as shown in Figure 2.20. The tiling factors (TC_o, TC_i, TH_o, TW_o) determine the size of the innermost loops and, therefore, the GLB size. In contrast, the permutations of the outermost loops determine the off-chip memory accesses and the data reuse.

2.3 Full precision vs quantized implementations

One of the main obstacles to deploying 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 used techniques for reducing memory and computation requirements is quantization. Quantization consists of mapping values from a continuous domain to a discrete and smaller one 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 called quantization error e_q (Eq. 2.1).

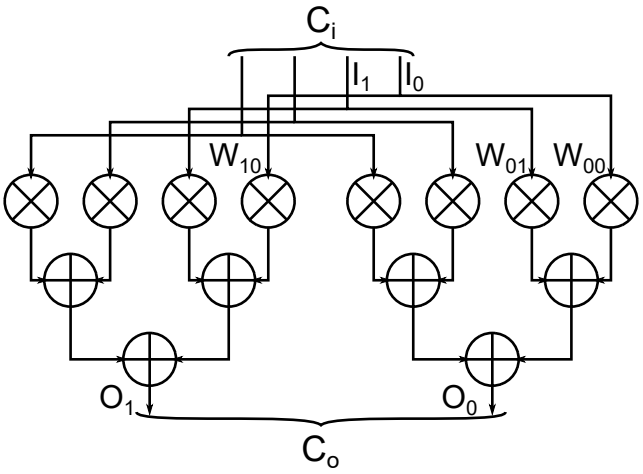


Fig. 2.19 No local reuse dataflow. [1]

$$e_q = x_q - x \tag{2.1}$$

From a hardware standpoint, quantization reduces the precision of the values and, consequently, the number of bits required to represent them. it is then possible to replace the floating-point representation with a fixed-point (see Figure 2.21). According to 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 change, leading to different ranges and different precision, making the fixed-point representation *dynamic*. This representation is beneficial for NNs, as weights and activations fall in several numerical ranges depending on the layer.

For example, a MAC executed on an 8-bit fixed-point number consumes 20x less energy than a MAC on a 32-bit floating-point number [88]; thus fixed-point representation allows for both memory and energy saving. Moreover, a number represented on 8 bits has a memory footprint of 4x smaller than one on 32 bits.

NN quantization aims to downsize the models, obtaining a lower memory footprint and a lower energy cost for both the computations and memory accesses.

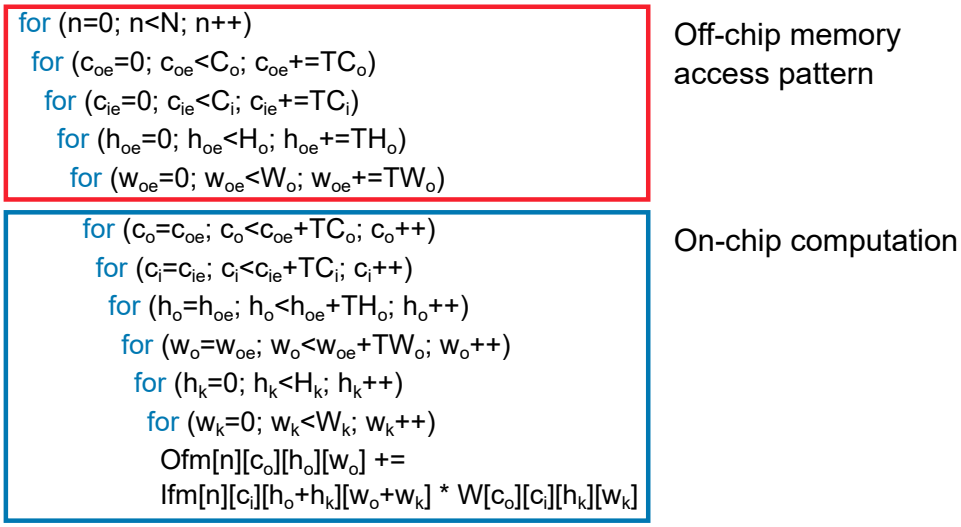


Fig. 2.20 Loop tiling technique applied to the 7-nested loops representation of the Conv layer. The outermost loops describe off-chip memory accesses, and the innermost loops determine the dataflow. [1]

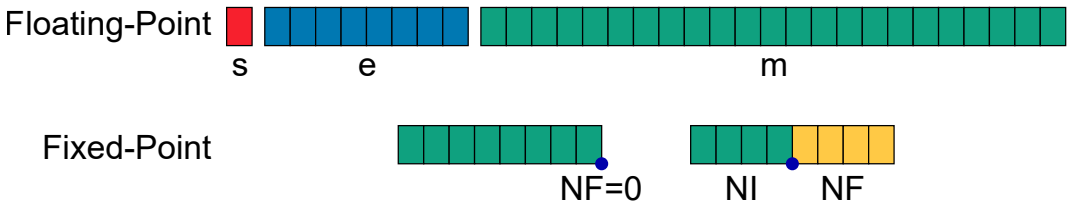


Fig. 2.21 **(top)** 32-bit floating-point representation. **(bottom)** 8-bit dynamic fixed-point representation. [1]

Nevertheless, the quantization must be applied with care in order not to reduce the accuracy of the models.

In NNs, three sets of values can be quantized: the weights, the activations, and the gradients. Many works focused on quantizing NN weights only since they directly impact the memory requirements [89][90]. While activations can be quantized at each algorithm's inference, the weights can only be quantized offline after the training phase, leading to two advantages:

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

Since the quantization of the activations affects the memory footprint and bandwidth as well as directly impacts the required computational resources, recent works focused on this topic [91][92][93].

The study of gradient quantization presents some limitations, mainly for two reasons:

- The NN training process is very sensitive to even minor variations in weight values; therefore, there is a risk of not achieving convergence. The quantization of the weights is a complex task.
- Usually, the training is performed only once offline on a GPU or a CPU, not on a resource-limited device, so there is no reason to devote too much effort to this quantization type.

Many different quantization methods have been investigated [94]. In the following, we will provide a summary of hardware-friendly quantization techniques, discerning between linear and non-linear methods (see Figure 2.22).

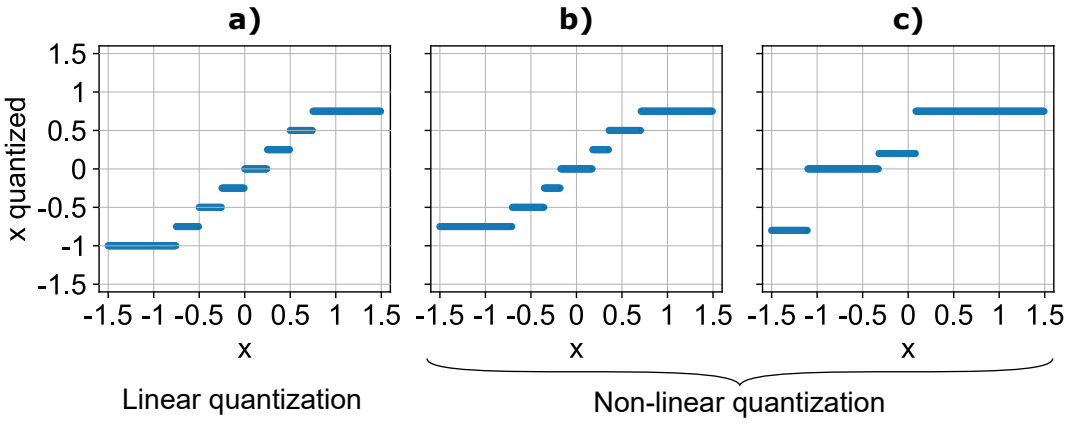


Fig. 2.22 Linear, logarithmic and vector quantization techniques. [1]

Linear Quantization: It is denoted by evenly-spaced quantization intervals, as shown in Figure 2.22.a. An example of linear quantization is represented by fixed-point coding, which, since its hardware implementation is well known, has been widely employed for NNs. Several works demonstrated that both weights and activations could be quantized to 8-bit dynamic fixed-point numbers with a negligible accuracy loss [95] [91]. It is possible to identify the quantization parameters (bitwidth and scale factor) by performing statistical analysis of the weights

and activations. Fine-tuning weights is then fundamental through a re-training step. The Ristretto framework [95] for example, can quantize complex models such as AlexNet [47], SqueezeNet [96] or GoogleNet [3] on 8 bits with less than 1% accuracy loss. Considering the great variety of layers present in an NN, using a different precision across the model may be helpful, i.e., layer-wise bitwidth. Works in [79] and [97] demonstrate that the bitwidth depends on the model depth, making a per-layer optimization of weights and activations bitwidth. In particular, in [97], it is stated that the weights bitwidth can decrease approaching the last layers of the NN, while the activations bitwidth remains constant.

Non-Linear Quantization: Generally speaking, weights and activations in a DNN model assume non-uniform distributions. Consequently, a non-linear quantization, where the quantization intervals are unevenly spread, can better fit their distributions as shown in Figure 2.22.b and 2.22.c.

Examples of non-linear quantization are logarithmic [98] and vector [99] quantization.

2.4 Activations and Weights Sparsity: Strategies and Encoding

Recent studies showed that most DNNs present inherent redundancy concerning the weights. Therefore, it is possible to prune them, i.e., pruning weights results in zero values that make the matrices sparse, without impacting the accuracy as presented in [100] and [101]. Both works demonstrate that the synapses can be reduced to percentages ranging from 20% to 80%, depending on the considered layers.

On the other hand, the ReLU, one of the most used activation functions, clamps negative activation values to zero during inference. Consequently, the null activations range from about 50% to 70%. Hence, these represent the primary sources for sparse matrices for activations and weights. Sparsity represents an excellent opportunity to optimize the inference phase:

- The DNN essential operation is the MAC between weights and activations. However, when one of the two is zero, there is no reason to perform the

operation, since the result will be zero. Therefore, it is possible to skip such operations to speed up execution and save energy.

- By employing compression techniques, it is possible to store 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 significantly cutting off the off-chip DRAM's access.

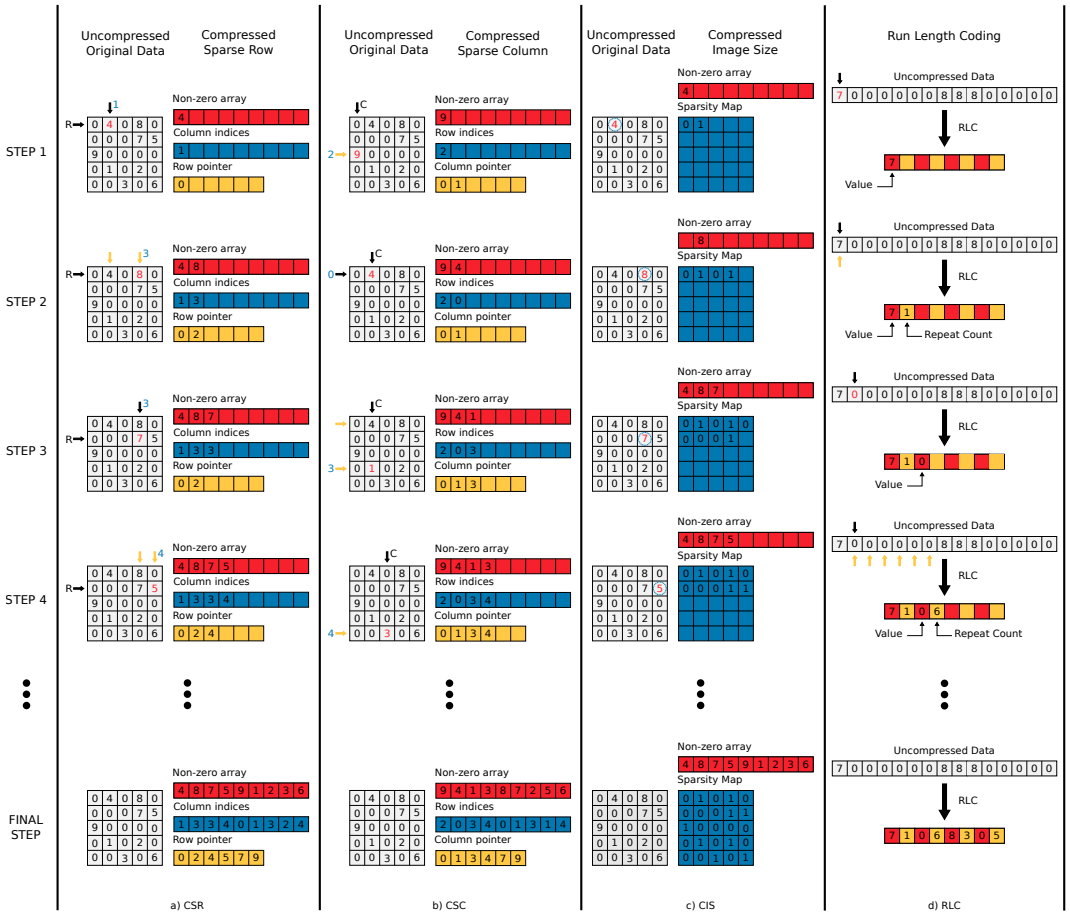


Fig. 2.23 Step-by-step compression formats comparison: (a) Compressed Spare Row (CSR), (b) Compressed Sparse Column (CSC), (c) Compressed Image Size (CIS), (d) Run Length Coding (RLC). [1]

Many data compression techniques can be found in the literature, however only a few hardware-friendly methods are used. Such techniques are composed of two sets of data: the first consists of non-zero values, while the second represents the metadata or the matrix indices necessary to reconstruct the original data pattern.

Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) are two formats belonging to the class of compressed stripe storage [102].

Compressed Sparse Row (CSR): As shown in Figure 2.23(a), the non-zero array stores all the non-zero values of the sparse rows in order, and an integer array (Column indices) stashes the related column indices. The row pointer stores the offsets within the previous two vectors, incrementally indicating the number of non-null elements per row. The number of bits required for such a representation is given as:

$$I \cdot (Nb \cdot (1 - Sp) + Ni \cdot (1 - Sp)) + (H + 1) \cdot No \tag{2.2}$$

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.

Compressed Sparse Column (CSC): CSC works like CSR, but considering data per columns. A non-zero array stores all the non-zero values of the sparse columns in order, and an integer array (Row indices) stashes the related row indices. The column pointer stores the offset within the previous two vectors, incrementally indicating the number of non-null elements per column. Figure 2.23(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.,:

$$I \cdot (Nb \cdot (1 - Sp) + Ni \cdot (1 - Sp)) + (W + 1) \cdot No \tag{2.3}$$

where W is the width of the input matrix

Compressed Image Size (CIS): This data format is composed of a sparsity map and a non-zero value list, as illustrated in Figure 2.23(c). The former is a mask with the same shape as the original data (1D vector, 2D Matrix or 3D matrices array) containing one bit per entry. The bit is zero if the corresponding value is null, one otherwise. The latter is an array composed of all non-zero values. In this case, the number of required bits has a simpler equation, as given below:

$$I \cdot (Nb \cdot (1 - Sp) + n) \tag{2.4}$$

where n is typically 1 bit.

Run Length Coding (RLC). It is a straightforward data format that can compress consecutive repetition of the same value as shown in Figure 2.23(d). In the case of sparsity, it is mainly used to compress consecutive zeros in a single zero and their count. Although it is straightforward to implement, it is only effective when zeros manifest compactly and consecutively (high percentage of sparsity). In addition, the RLC is designed for data arrays, so it is suboptimal when operating on matrices.

Figure 2.24 compares the compression ratio of CSR, CSC, CIS and RLC methods. The compression ratio is the relationship between the compressed bit size and the original model. The picture shows two boundaries for the four 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. It is possible to notice that the CIS format outperforms the CSR or CSC formats in almost all the sparsity ranges.

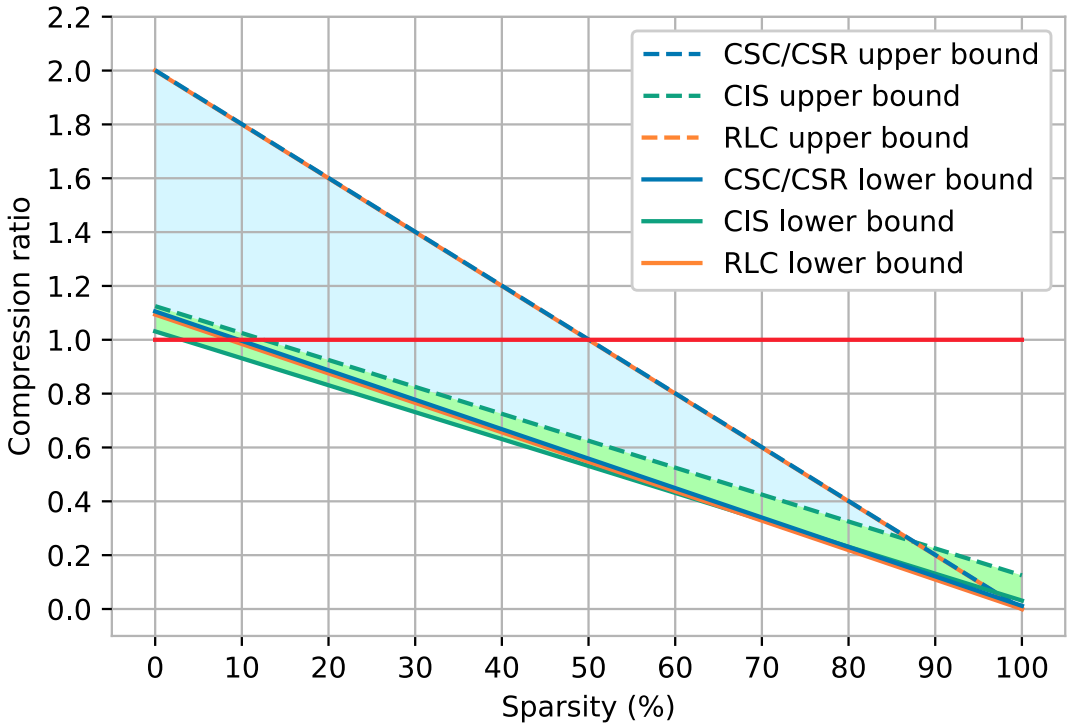


Fig. 2.24 Compressed ratio with variation in sparsity of CSC, CSR, CIS and RLC. The picture includes two boundaries: the upper bound refers to Conv 4 of AlexNet ($3 \times 3 \times 384$) with data parallelism of 8 bits, while the lower bound is based on Conv 1 ($11 \times 11 \times 3$) represented on 32 bits. [1]

2.5 Conclusions

The importance of the architectural exploration described in this chapter lies in the ability to exploit a given feature (or features) of the network model or the data itself to derive an optimized strategy. For example, the structure of a neural network, or better of its layers, when decomposed, can lead to several data flow choices that, in turn, allow the design of multiple possible digital architectures. Similarly, exploiting a given data characteristic such as sparsity allows for accelerated execution if all the operations that do not contribute to the final result are discarded. To conclude, starting from a domain-specific application, it is possible to derive several data flows and data properties considerations that, through architectural exploration, can lead to highly optimized hardware solutions in terms of computation and power efficiency. This approach is critical in the edge computing scenario, where the resources are extremely limited.

Chapter 3

Enabling the Training on the Edge: an alternative to Backpropagation

3.1 Introduction

Performing training directly on the end device will require no (or negligible) data transfer. Data collected by sensors are used locally and nothing has to leave the device. Besides a considerable energy saving, not having a direct and continuous data stream towards the cloud, it would also significantly increase privacy and security. The data should no longer be encrypted and there is no risk that they will be intercepted in transmission. In conclusion this is crucial for all areas requiring anonymized datasets or that process sensitive data such as medical applications. As the data have neither to be transmitted nor received back, latency could be significantly reduced. For example, Google [103] has estimated that an autonomous vehicle which generates 3TB of information per hour would require almost 9 hours with a bandwidth of 1Gbps (maximum 5G speed) to upload the data to the cloud.

Having individual devices that can learn locally would be a revolution for the IoT, but also for the DL in general. Starting from a pre-trained model or one without any notion, it would be possible to get highly specialized applications adapted to the environment in which they are inserted. The end-users could take advantage of applications tailored to their needs and usage.

From the point of view of basic operations, inference and backpropagation-based training have no differences. Both processes require convolutions between matrices. However, while in inference, the neural network model is traversed only once forward to obtain prediction, in training, the network is traversed both forward and backward, requiring almost twice as much computation. Besides, while inference can be performed even on a single sample, backpropagation typically uses batches to make the process faster and more effective. Working on batches complicates operations as the amount of data to be processed and stored increases significantly the number of operations required and the power required to process it. These represent two significant limitations for the training on the Edge.

Increasingly complex network models are born, in tandem with cutting-edge hardware platforms. Progressively efficient devices have allowed the DNN inference to reach the most extreme nodes of the IoT, automatically becoming part of edge computing. However, training remains the exclusive subject of cloud servers, with the exception of some distributed learning approaches, federated Learning [104], across multiple IoT nodes. In this scenario, the attractiveness and the need for an approach to transfer training on edge grows, for a unique and complete device capable of executing and training DL algorithms. The training can be moved, at least partially, on the final device allowing a continuous and slow learning in order to improve the algorithm. In addition, we may think of devices that can learn new tasks (new classes, in case of classification), without having to retrain the full network.

This chapter aims to propose an alternative technique to BackPropagation, namely Extreme Learning Machine (ELM), applied in a Continual Learning (CL) scenario in which new tasks must be learned without forgetting previous ones. By adapting ELM to CL, cutting-edge results comparable with state-of-the-art can be achieved. The following chapter consists of an introduction to CL and related benchmarks. Then the ELM is introduced along with a variant of it. The latter technique is finally combined and adapted for CL tasks. Finally, the results and performance are analyzed.

To be noted that this thesis work proposes a methodology for low-power on-the-edge continual learning, but does not address any specific experimental application.

3.2 Continual Learning (CL) paradigm, the road towards on-chip learning

3.2.1 Background

Continual Learning (CL) [105][106][107], is an algorithm capable of learning from continuous flows of information with a number of non-predefined tasks. CL techniques update the models by exploiting only the latest data. This appears to be the only realistic solution when learning on the edge from massive streaming or ephemeral data. Indeed, it would be unfeasible to keep in memory all the previous data and process them from scratch every time a new sample of information becomes available. This approach could provide a very flexible algorithm to learn with minimal memory and power requirements. CL introduces three crucial challenges: catastrophic forgetting, application scenarios, and benchmarking, as discussed in the following.

The catastrophic forgetting [108][109][110] occurs when the learning process radically changes the weights of the network, making them forget previously learned tasks. To overcome this problem several techniques have been proposed in literature:

- **Architectural strategies**. They consist of specific layer architectures or weight freezing approaches aimed at mitigating catastrophic forgetting (Copy Weight with Re-init CWR [11], AR1 [111]).
- **Regularization strategies**. They extend the loss function with terms that encourage the consolidation of a selection of weights which are important to preserve (Learning Without Forgetting LWF [112], Synaptic Intelligence SI [113], AR1 [111], ICARL [114]).
- **Rehearsal strategies**. The model is periodically re-trained on past data in order to consolidate some of the memory connections already learned. These approaches require to store previous samples (ICARL [114]).

Some methods combine differnt techniques to achieve higher performance. For example, AR1 [111] is a combination of Copy Weight with Reinit (architectural strategy) and Synaptic Intelligence (regularization strategy). AR1 combines the architectural strategy, which implicates freezing part of the network and training a

small subset of weights, and the regularization strategy employed to adjust the loss function that selects the proper weights to update during training. More in detail, it combines the features of CWR [11] and SI [113]. In CWR, the learned weights of the classifier are saved in a consolidated database. After the training for adding new classes, the old weights are loaded back so that the old tasks are not affected by the new ones. However, the weight values must be scaled to maintain balance among tasks since the size of subsequent batches may differ. In CWR, this scaling is performed considering the weights of a specific class inside the batch. At the same time, AR1 proposes an optimization that consists in subtracting the average of all weights in the classifier from each consolidated weight. Moreover, to improve performances, AR1 does not freeze the CNN, it applies some fine-tuning, plus it implements the regularization techniques found in EWC and SI. They are based on weight "importance", which indicates how much the loss function is sensitive to small changes with respect to a certain weight in the network. When fine-tuning is applied, high-importance weights must not undergo significant variations.

Considering the application scenarios, most studies are based on a Multi-Task (MT) scenario, where the model is required to learn several isolated assignments incrementally without forgetting the previous ones. An example could be to split MNIST into five isolated tasks, two classes each, with no overlapping among different classes. Then, the accuracy is computed separately for each task, like in [113].

Even though these experiments are suited for accessing the feasibility of training a model on disjoint tasks, they do not adequately represent the incremental task problems. A compelling and unexplored scenario represented by the Single-Incremental-Task (SIT) is addressed in [114] and [11]. An example of SIT is the class-incremental learning, in which new classes are added sequentially. Generally speaking, SIT paradigm is more complex than MT one: it is still affected by catastrophic forgetting; the model has to learn to discern classes that it usually never encounters together. In MT, the output neurons are clustered into distinct classification layers (one for each task). SIT instead employs a single output layer that includes the neurons of all the classes learned so far. When training a model in an MT scenario, the output layer of the batch is trained separated while sharing the rest of the NN weights. On the other hand, in SIT, memorized weights could be exploited when learning the current batch classes. During MT assessment stage, each task can be evaluated individually with the related classification layer. Instead, in the SIT paradigm, the assessment is performed agnostically concerning the membership of

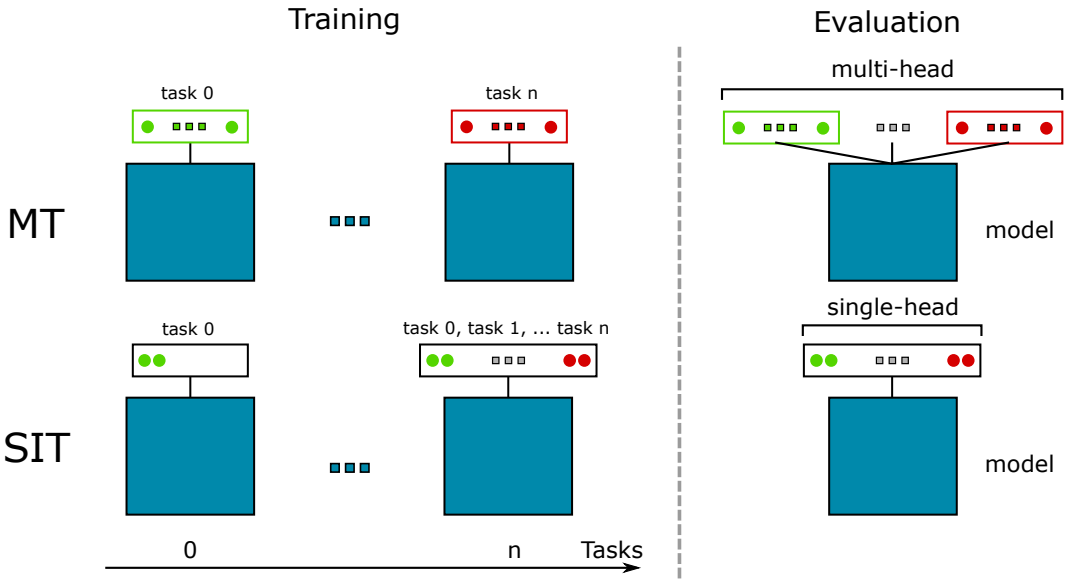


Fig. 3.1 Comparison between Multi-Task and Single-incremental-task scenarios for both training and evaluation stages.

an instance to an explicit batch. Thus final probabilities are sorted out by employing a unique softmax layer, as shown in Figure 3.1. This requires comparing samples that were never encountered together during the training phase and can profoundly affect final accuracy. In the CL context, the term "head" is generally used to denote the output classification layer, consequently the MT scenario implements multiple disjoint heads, while SIT features a single expanding head.

Moreover, in SIT there are three possible types of content update:

- **New Instances (NI)**: new training samples of the same classes become available in future batches due to new poses and environment conditions (brightness, background, rotation, etc.).
- **New Classes (NC)**: new training samples belonging to new classes become available in subsequent sets (class-incremental learning).
- **New Instances and Classes (NIC)**: unexplored training samples belonging to both known and new classes become available in subsequent training batches.

Table 3.1 MT and SIT current becnhmarking scenario.

Dataset	Multi_Task (MT)	Single-Incremental-Task (SIT)
Permuted MNIST	[115]	-
Rotated MNIST	[115]	-
MNIST Split	[113]	[107]
CIFAR 10/100 Split	[113]	-
CORe50	-	[11]

3.2.2 Benchmarking

Nowadays, benchmarking in CL is still extremely nonstandard as described in Table 3.1. For example, considering supervised classification, researchers often document their results on dissimilar datasets by exploiting different training and evaluation protocols. Many datasets lack of multiple views of the same object under different conditions such as lighting, pose, and variable background. For CL scenarios, datasets and benchmarks specifically designed are required to evaluate several emerging approaches. In addition, the ideal dataset should comprehend many classes split into subsequent batches presenting each class under different conditions. Another crucial feature is the presence of temporal coherency among samples , for instance, short videos in which the object is slightly moving. These features can improve classification accuracy and facilitate object detection [116]. All these features are encapsulated in CORe50 (Continuous Object Recognition). CORe50 is a dataset composed of 50 domestic objects belonging to 10 different categories: balls, cups, mobile phones, light bulbs, scissors, etc. (see Figure 3.2). Two types of classification can be performed: at the object level (50 classes) or the category level (10 classes).

In the following sections, the work will focus on SIT with NC update content type, representing one of the most challenging scenarios.



Fig. 3.2 Some samples from CORe50 [11]. CORe50 is a dataset consisting of 50 household objects divided into 10 different categories such as: balls, cups, cell phones, light bulbs, scissors, etc. Each object has multiple samples from different angles.

3.3 Extreme Learning (ELM), an alternative to classic Back-propagation

3.3.1 Background

Although backpropagation is a well-known and widely used process, training feed-forward networks requires in some cases hours of processing-time. This time is mainly due to two factors: the gradient-based training is generally very slow, the massive amount of parameters of a network must be tuned iteratively. The Extreme Learning Machine (ELM) represents an alternative learning approach with good generalization performance.The main difference between ELM and BP lies in the fact that the former tends to define all the weights in one step (as shown below), while BP requires thousands of iterations to refine the weights each time. However, one of the significant limitations of ELM is that it can only train a Single hidden Layer Feedforward neural Network (SLFN).

In [117], Huang et al. present an ELM-based method to train single-hidden layer feedforward neural networks capable of determining the weights analytically. This method is developed from two cornerstone principles, which are the minimum norm least-squares and the Moore-Penrose generalized inverse [118]. The proposed

algorithm aims to achieve the smallest training error, with the smallest norm of weights and excellent generalization performance.

Let us to have:

- A neural network with a single hidden layer with L neurons,
- $w_j = [w_{j1}, w_{j2}, ..., w_{jn}]^T$ as the weights connecting the jth hidden node and all the input nodes,
- $\beta_j = [\beta_{j1}, \beta_{j2}, ..., \beta_m]^T$ as the weights connecting the jth hidden node with the output nodes,
- b_j as the bias for the jth hidden node,
- $f(x)$ as the activation function for the nodes in the hidden layer.

Considering N input samples (x_i, t_i) in the dataset, $x_i = [x_{i1}, x_{i2}, ..., x_{in}]^T \in R^n$, $t_i = [t_{i1}, t_{i2}, ..., t_{im}]^T \in R^m$, where n is the number of input features, m the number of output classes, the network output can be modeled as follows:

$$\sum_{j=1}^{L} \beta_j f(w_j \cdot x_i + b_j) = y_i \qquad i = 1, ..., N \tag{3.1}$$

The objective is to find a combination of β_j, w_j and b_j such that the output y_i represents the target output t_i. Defining the hidden layer matrix H and the target matrix T as:

$$H = \begin{bmatrix} f(w_1 \cdot x_1 + b_1) & \dots & f(w_L \cdot x_i + b_L) \\ . & & . \\ . & \dots & . \\ . & & . \\ f(w_1 \cdot x_N + b_1) & \dots & f(w_L \cdot x_N + b_L) \end{bmatrix}_{N \times L}$$
$$T = \begin{bmatrix} t_1^T \\ . \\ . \\ . \\ t_N^T \end{bmatrix}_{N \times m} \tag{3.2}$$

Considering fixed input weight w_j and hidden biases b_j, training an SLFN means finding the least-squares solution of the linear system of 3.1 as

$$H\beta = T \tag{3.3}$$

If the number of hidden neurons L equals the number of distinct training examples, then the H matrix is square and invertible. Consequently, with w_j and b_j randomly chosen, the SFLN can be approximated with zero error. However, in some cases, L could be smaller than the number of distinct training examples. In this case, H is not invertible, and the smallest norm least-squares solution of the previous equation can be found by computing H as the Moore-Penrose generalized inverse.

In [117], the authors propose two possible solutions to determine β analytically according to the size of the batch N:

$$\begin{aligned} \beta &= H^T(\frac{I}{C} + HH^T)^{-1}T \quad N \leq L \\ \beta &= (\frac{I}{C} + HH^T)^{-1}H^T T \quad N \gg L \end{aligned} \tag{3.4}$$

where I is the identity matrix and C a regularization parameter. Thus, the ELM algorithm consists of four steps:

- random input weights w_j and biases b_j initialization, with $j = 0, 1, ..., L$;

- computation of the hidden layer matrix H;
- computation of the pseudo-inverse of H;
- output weights β computation.

This methodology allows us to train a network without iterations, representing a potential solution to speed up the learning process. However, it can only train a network consisting of a fully connected layer. This is a limitation for more complex tasks that require advanced classifiers or convolutional networks.

3.3.2 Online Sequential-Extreme Learning Machine (OS-ELM): ELM with batches

In [119], the authors propose the Online Sequential-Extreme Learning Machine (OS-ELM), capable of training the network with data given in batches even composed of a single sample. The idea is to divide the H and T matrices into multiple parts and compute the output weights incrementally.

$$H = \begin{bmatrix} H_0 \\ H_1 \end{bmatrix} \quad T = \begin{bmatrix} T_0 \\ T_1 \end{bmatrix} \tag{3.5}$$

Let's consider that in the previous step we computed the output weight for the first batch H_0 and T_0 :

$$\beta_0 = (H_0^T H_0)^{-1} H_0^T T_0 \tag{3.6}$$

To update the weights with the next batch we should compute:

$$\beta_1 = \left(\begin{bmatrix} H_0 \\ H_1 \end{bmatrix}^T \begin{bmatrix} H_0 \\ H_1 \end{bmatrix} \right)^{-1} \begin{bmatrix} H_0 \\ H_1 \end{bmatrix}^T \begin{bmatrix} T_0 \\ T_1 \end{bmatrix} \tag{3.7}$$

However, it is fundamental to find another way to compute β_1, without using the previous batches H_0 and T_0. This is possible by relying on β_0 and the new batch only. In [119], the authors propose the following:

$$\beta_0 = \beta_1 + K_1^T H_1^T (T_1 - H_1 \beta_0) \tag{3.8}$$

where

$$K_1 = [H_0^T \; H_1^T] \begin{bmatrix} H_0 \\ H_1 \end{bmatrix} = K_0 + H_1^T H_1 \tag{3.9}$$

Let's suppose that we already computed β_k for k batches and that the $(k+1)$ batch is received, the next weight would be updated as:

$$\begin{aligned} P_{k+1} &= P_k - P_k H_{k+1}^T (I + H_{k+1} P_k H_{k+1}^T)^{-1} H_{k+1} P_k \\ \beta_{k+1} &= \beta_k + P_k H_{k+1}^T (T_{k+1} - H_{k+1} \beta_k) \end{aligned} \tag{3.10}$$

where:

$$P_{k+1} = K_{k+1}^{-1} = (K_k + H_{k+1}^T H_{k+1})^{-1} \tag{3.11}$$

In this way it is possible to compute the output weight for the new batch without the need for the previous data. However, it is important to notice that the matrix P_k of size $[L \times L]$, computed in the previous step, must be retained in memory for the next training phases.

There exist another possible implementation for computing the output weights without retaining the previous batches. Let's consider the product $H^T H$, dividing H into different batches we obtain:

$$H^T H = [H_0^T \; H_1^T \; \dots \; H_k^T] \begin{bmatrix} H_0 \\ H_1 \\ \cdot \\ \cdot \\ \cdot \\ H_k \end{bmatrix} = H_0^T H_0 + H_1^T H_1 + \cdots + H_k^T H_k \tag{3.12}$$

This means that the matrix multiplication can be obtained adding the contribution of each batch separately, this also applies to the $H^T T$ product. Thus, the output weight can be computed as:

$$\beta = \left(\sum_{i=1}^{k} H_i^T H_i + \frac{I}{C} \right)^{-1} \left(\sum_{i=1}^{k} H_i^T T_i \right) \tag{3.13}$$

Let's consider that we received k batches and for each of them we computed the two products $H^T H$ and $H^T T$ and we accumulated their results in the matrices M and P, respectively. Then, in order to compute the output weights, all we need to do is

compute the inverse M and multiply it with P:

$$\beta_k = (M + H_k^T H_k)^{-1}(P + H_k^T T_k) \tag{3.14}$$

This means that the matrices M and P require to be stored in memory and extracted during the training phase. The results and performance are similar to the OS-ELM except for some approximations in the computation. The dimension of M and P are, respectively, $[L \times L]$ and $[L \times m]$, where L is the number of the hidden nodes and m the number of the output classes. This technique is referred to as Parallel Regularized - Extreme Learning Machine (P-ELM) [120]. It is used for parallel computation, meaning that the computational load can be divided among different devices. Each device can elaborate the two matrices M and P and return them to a central station which gathers all the results.

3.4 Contribution and key idea: ELM for Single Incremental Task scenarios

3.4.1 ELM and OS-ELM: evaluation and comparison with Backpropagation

In order to investigate the features of the ELM algorithms, some basic tests were performed taking MNIST split and CIFAR10 split as references. ELM is very simple as an approach, however, it allows for training only single-layer feedforward networks (fully connected). Consequently, as a first approach to see the actual learning capability and speed of execution, we considered fully connected networks with 10 neurons on the output layer (classifier) representing the classes of the two datasets and an increasing number of neurons in the hidden layer. Initial weights are randomized with a uniform distribution between -1 and 1. All the trainings are performed using an Intel Core i7-7700HQ CPU (2.80 GHz).

Figure 3.3 compares different training performed on MNIST split and CIFAR10 split. Considering the accuracy, we can notice that regarding the first dataset (column of left), the model successfully reached a discreet accuracy (greater than 95%) with just 2000 hidden nodes. This is mainly because the MNIST dataset is quite simple, and a fully connected layer is sufficient to learn its content, also the model

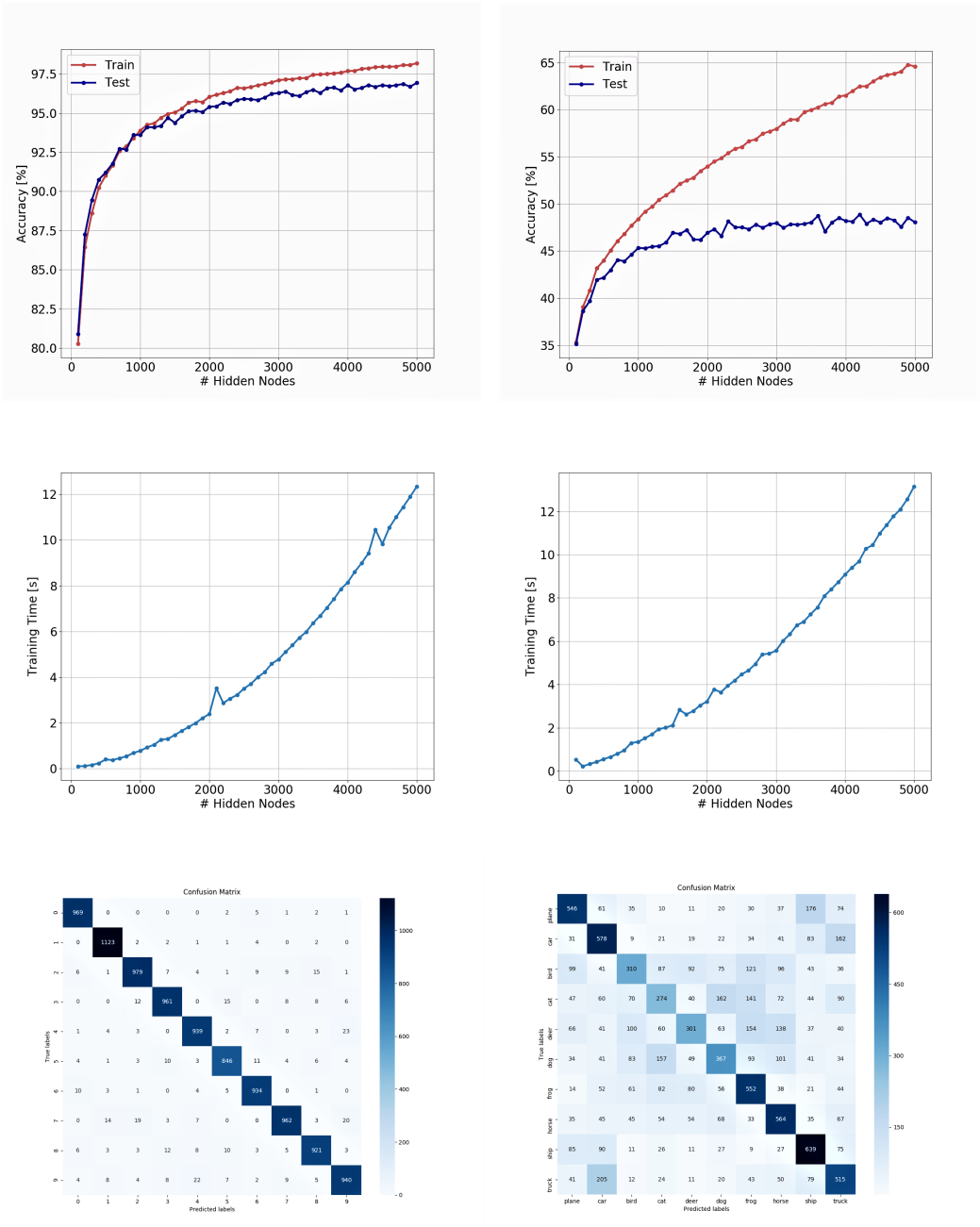


Fig. 3.3 ELM training performance on MINIST split (left column) and CIFAR10 split (right column).

Table 3.2 Training latency for BP and ELM on different devices.

Device	**Backpropagation**	**ELM**	**Speed up**
NVIDIA Jetson TX2	10.2 s	0.6 s	17x
Intel Core i7-7700HQ	74.4 s	3.90 s	19x
NVIDIA GTX 1050	8 s	0.5 s	16x
NVIDIA Tesla K40	1.3 s	0.1 s	13x

can generalize quite well. On the contrary, for CIFAR10, the model can follow the training pleasingly as the number of hidden nodes increases but unfortunately can not generalize the problem good enough (overfitting). This is mainly due to two reasons: this dataset is more complex than the first and requires the extraction of many more features, so it would be necessary to add one or more hidden layers, second would be helpful to add some regularization techniques during training (dropout). This thesis is further corroborated by the confusion matrices (third line in the figure) where it is clearly shown that while for MNIST the confusion matrix is exceptionally clean, for CIFAR10 some classes are classified incorrectly. In particular, the "truck" is often misclassified with "car", but also "deer" and "frog" are often mistaken. For both datasets, after 2000 hidden nodes, the accuracy no longer improves with enough slope to justify the increase in neurons, so ideally, a model of that size should be used.

Considering the training time it is possible to notice some interesting features. Considering 2000 hidden nodes, it is possible to notice that to learn the entire MNIST dataset the ELM requires 2.1 seconds, while CIFAR10 takes 3.80 seconds. This difference is due to the sample size of the datasets. MNIST uses 28×28 grayscale images, while CIFAR10 uses 32×32 RGB images. However, with the same accuracy, Back Propagation requires a time of 74.35 seconds to train 2000 hidden nodes on MNIST with a learning rate of 0.01, momentum equal to 0.9 and weight decay to $5\cdot10^{-4}$. This means that the ELM increases the traning speed by $19.6\times$. Table 3.2 compares the latency of BP and ELM algorithms when training same model (2000 hidden nodes SFNL) on different devices. ELM can perform up to $19\times$ faster.

As explained earlier, the OS-ELM and P-ELM allow for a model that can continuously learn as new data become available. In continuous learning, or specifically in an edge computing context where new data may be available at different times and in different quantities, the ability to accommodate batches of different sizes is essential. One of the main problems in the training phase is that the batches

must have a consistent size for good optimization, generalization, and to allow very deep networks to converge. This has been particularly noted in the case of Batch Normalization, considered a cornerstone of many state-of-the-art computer vision algorithms. BN suffers from small batch sizes showing an increasing error due to inaccurate batch statistics estimation [121].

Table 3.3 OS-ELM training test of a SFLN on CIFAR10 with different batch sizes.

Batch size	**TRAIN accuracy**	**TEST accuracy**	**Training time (single batch)**	**Training time (entire dataset)**
50,000	58.05%	47.85%	6.49 s	6.49 s
5,000	58.05%	47.85%	1.08 s	10.8 s
1,000	58.05%	47.85%	0.44 s	22 s
1	58.05%	47.85%	0.24 s	12000 s

In order to verify if the batch size influences the training process in terms of accuracy, some training with variable batch size was considered. OS-ELM was used to perform training of an SFLN network on CIFAR10. The results reported in Table 3.3 show that the accuracy never varies with the batch size, this shows that ELM can always find the minimum cost function. Furthermore, considering that each training had its weights randomization and the dataset was shuffled beforehand, it can be deduced that even the order of the data does not matter, i.e. no matter which or how many batches the network has to process, at the end of the time the result will always be optimal. This is very important, especially in CL scenarios where batches may be very unbalanced in terms of samples, both in quantity and distribution.

The training time per batch reduces as the batch size decreases. This is due to the fact that all matrix sizes (except P and M) are reduced, making the computation much faster. However, since smaller batches mean more batches, the total training time is higher because the number of multiplications and transpositions to perform are more.

Figure 3.4 compares the training time for OS-ELM and P-ELM for different batch sizes. Two networks are considered, one with 2500 nodes (left), and one with 5000 nodes (right). OS-ELM tends to be faster for small batches, since, even if it presents higher computation complexity with respect to the P-ELM algorithm, its inverse depends on the batch size. On the contrary, the P-ELM inverse depends on the number of hidden nodes. Considering that in the case of CL applied to edge

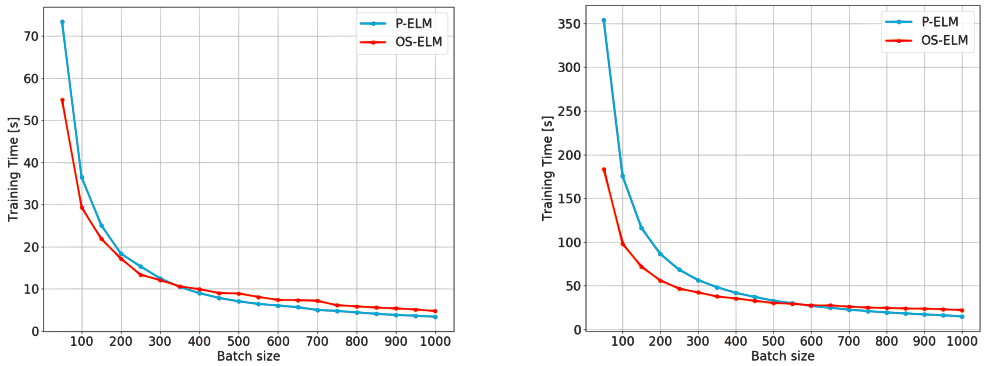


Fig. 3.4 P-ELM vs. OS-ELM comparison in terms of time per batch size.

computing the batch size is small (few samples), OS-ELM would seem to be the best option. So from here on, considering a CL-based scenario, we will rely on OS-ELM.

3.4.2 Key Idea: OS-ELM for SIT

Considering a convolutional network, the last layer is always fully connected, thus, using a pre-trained network, it is possible to exploit ELM to train only the final classifier. Convolutional layers provide input-extracted features to the last layer coached with ELM that provides classification. This approach can be applied to a generic network that can be instantly trained with ELM and therefore specialized on the fly.

In addition, the last classification layer can continuously be trained at runtime as new samples are available, allowing an on-the-edge training, while the whole network may undergo a fine-tuning once in a while to improve the entire classification. This can be perfectly integrated with Continual Learning. However, to avoid catastrophic forgetting, the ELM requires to save all the previous samples. This is impossible as one would need an ideally infinite memory and a considerable computing capacity to reverse a matrix gradually increasing in size. In fact, in most cases, the number of neurons in the output layer is known a priori and strictly depends on the target dataset. However, in the case of CL, the classifier could require a reshape since new unknown classes could be encountered. In this case, the classifier has a specific initial size that increases over time as new batches and samples become available, such as the SIT case.

However, using OS-ELM can avoid catastrophic forgetting and constantly update the classifier, only storing M and P matrices. This keeps track of all the experienced samples without storing them. In addition, as demonstrated earlier, the classifier can learn regardless of the size and order of the batches encountered. This feature gives the OS-ELM outstanding flexibility and advantage in a scenario such as CL. This work combines the OS-ELM algorithm with Continuous Learning for the Single Incremental Task applications to reach state-of-the-art results. In the following sections, we first introduce the experimental setup regarding the tests performed and then present the accuracy and performance analysis.

3.4.3 Experimental Set-up

In order to have a fair comparison, the Continuous Objects Recognition (CORe50) dataset is used as a reference. It provides 164.866 RGB images of size 128x128 pixels. Here we target NC content update type where the 50 classes are clustered in 9 batches delivered sequentially: B1 includes 10 classes while B2, . . . , B9 5 classes each. Each class provides 2400 samples (300 frames × 9 sessions) for the training batches and 900 patterns (300 frames × 3 sessions) for the test set. The test set is stationary and the accuracy is assessed after each encountered batch on the entire test set, including unobserved classes.

Table 3.4 Summary of network adjustments to accomodate OS-ELM.

CaffeNet		
Layer	**Original**	**This work**
Input	size: 227x227	size: 128x128
conv1	stride: 4	stride: 2
conv2	pad: 2	pad: 1
fc6	nodes: 4096	nodes: 4096 (re-init)
fc7	nodes: 4096	nodes: 50
fc8	nodes: 1000	removed
GoogLeNet		
Layer	**Original**	**This work**
Input	size: 227x227	size: 128x128
conv1	stride: 2, padding: 3	stride: 1, padding: 0
classifier	nodes: 1000	removed

Two neural networks are used as reference models: CaffeNet and GoogLeNet. Both models have been designed to target the ImageNet dataset, so their input sizes are 227x227 and 224x224, respectively. However, CORe50 provides images of size 128x128, thus the stride and padding are appropriately reduced. In addition, other adjustments are needed, especially in the last layer, the classifier, as reported in Table 3.4.

In Caffenet, the fully-connected layer fc6 is substituted with one of the exact sizes with random re-init weights, while fc7 goes from 4096 neurons to 50, and the fc8 is removed. In GoogleNet, the modifications are more complex. GoogLeNet presents two auxiliary output layers used exclusively during the training phase, removed for the AR1 and OS-ELM. In addition, the classifier is adjusted accordingly. In fact, as mentioned, to apply ELM on a CNN the classifier must be substituted with one hidden layer and the output layer with 50 neurons.

3.4.4 Accuracy Evaluation

The results obtained with CaffeNet and GoogLeNet are compared with the results of AR1 [111] and shown in Figure 3.5. The Cumulative and Naive methods represent two extreme approaches. The cumulative consists of storing all the encountered batches and training the model from scratch at each step by using BP. It is considered the upper bound limit in term of accuracy. On the other hand, the Naive approach updates the network through BP without any method to prevent catastrophic forgetting. As it is possible to notice, this is the worst-case and depicts how forgetting can affect a model.

The ImageNet pre-trained models are used for both CaffeNet and GoogLeNet, reaching an elevated level of complexity in the extracted features. The results obtained for the CaffeNet model are shown in Figure 3.5 left. It can be noticed that the accuracy (about 13%) for the first batch is lower than the one obtained in AR1 (20%), as in AR1, all the CNN undergo a fine-tuning process. Considering the encountered batches, the trend follows a similar pattern both for OS-ELM and AR1. After the last batch, the OS-ELM accuracy reaches the 47%, which is higher than the AR1.

The same test is carried out for GoogLeNet, the results are shown in Figure 3.5 right. As can be noticed, the ELM accuracy (about 50%) did not reach the AR1 one.

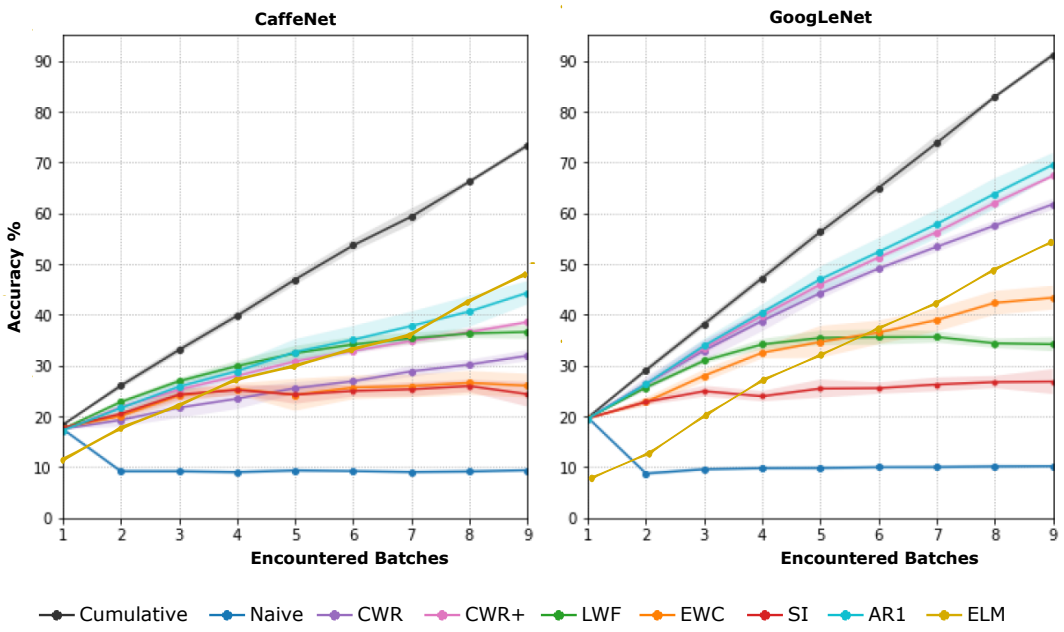


Fig. 3.5 Results comparison for CL NC performed on CaffeNet (left) and GoogLeNet (right).

A potential reason is that this network is far more complex than the previous one and, training just the classifier layer could not be enough to obtain good generalization. Therefore, we tuned the generalization hyperparameter C to find the best accuracy possible. Regarding the NIC scenario, there are no data available for AR1. In this case, we compared the results obtained using ELM with the CWR algorithm. The results for ELM are equivalent to the one obtained for the NC scenario since the ELM is not sensitive to the order in which the batches are provided and the number of batches into which the dataset is split. The accuracy shows better results concerning the CWR algorithm.

3.4.5 Performance Evaluation

Besides accuracy, it is essential to understand the OS-ELM algorithm's memory requirements and computational complexity.

Non-temporary elements to be stored:

- CNN Weights and biases

Table 3.5 ELM computation complexity.

Operation	Complexity
$H^T H$	$O(hn^2 \cdot bs)$
$M + (H^T H)$	$O(hn^2)$
$H^T H$	$O(hn \cdot bs \cdot c))$
$P + (H^T T)$	$O(hn \cdot c)$
M^{-1}	$O(hn^3)$
$M^{-1} \cdot P$	$O(hn^2 \cdot c)$

- ELM weights, matrix of size $hn \times in$, where hn is the number of neurons in the hidden layer, while in is the number of neurons in input to the classifier.
- Biases present in the ELM hidden layer, vector of size hn.
- ELM output weights (β), matrix of size $c \times hn$, where c is the classifier.
- Matrix M: matrix of size $hn \times hn$, which stores $H^T H$.
- Matrix P of size $hn \times c$, which stores $H^T T$.

Temporary elements to be saved during execution:

- Matrix X of size $bs \times d \times h \times w$, contains the input samples, where bs is the batch size, d, h and w are the image depth, height and width, respectively.
- Matrix T of size $bs \times c$, contains the input samples labels.
- Matrix H of size $bs \times hn$, contains the extracted features of the input batch.

The fundamental matrices for ELM are M and P which contain the data related to the previously encountered batches. AR1 algorithm requires storing a temporary matrix containing the output weights of the last layer, and two times a matrix of the size of weights and biases of the entire network. The first one stores the actual weights and biases, while the second stores the importance of each element. Considering CaffeNet the memory requirements for both algorithms would be:

$$AR1 = 2(wb + (in \cdot fc1) + (fc1 \cdot fc2)) + c \cdot hn \tag{3.15}$$

$$ELM = wb + (in \cdot hn) + (hn \cdot hn) + 2(c \cdot hn) \tag{3.16}$$

These two equations are comparable and present the same order of magnitude (about 60 M parameters). Concerning computation, Table 3.5 shows the complexity of each operation. As can be seen, the inverse has the most complex operation.

3.5 Conclusions

In this chapter, we combined the features of the Extreme Learning Machine with Continual Learning. In particular, after carefully analyzing ELM and its variants (OS-ELM and PE-ELM), we adopted the Online Sequential ELM to solve the challenges proposed by CL in the Single Incremental Task scenario. The SIT requires the neural network to be able to continuously learn as new batches of samples become available over time. This not only implies that the network weights must be continuously updated without affecting what has been learned up to that point, but also requires that the final layer, operating as a classifier, is expanded whenever a new class is encountered.

In order to obtain fair comparisons with state-of-the-art, benchmarking was defined by considering two networks such as CaffeNet and GoogLeNet trained in SIT on CORe50. The two network models were suitably modified in the last layers to allow the inclusion of the OS-ELM as shown in Table 3.4. This approach shows encouraging results, for instance, in the case of CaffeNet our technique outperformed AR1, the current state-of-the-art in accuracy, reaching an accuracy of 47%. In contrast, considering GoogLeNet, a network of greater complexity than the former, our technique cannot go beyond AR1. Consequently, we can state that OS-ELM is an excellent technique for CL in SIT, however, for complex networks, it is not enough to operate only on the classifier, but it is necessary to include, at least partial, fine-tuning of the network.

OS-ELM features two essential skills that should be considered in addition to the accuracy it can achieve. In fact, its flexibility allows it to learn from batches of any size, even with single-element batches, which is critical for CL. Moreover, regardless of the order of the batches encountered, the result in accuracy will always be the same.

Chapter 4

A Multi-Precision Bit-Serial Hardware Accelerator IP for IoT

As described in the previous chapter, architectural exploration is a fundamental practice in hardware architecture design. Moving through the space of possible solutions allows identifying the best trade-off between the metrics considered (area, power, latency, etc.) and optimizing the final solution. A general-purpose solution, e.g., microcontrollers and microprocessors, provides high flexibility, wherein a vast range of applications can be executed, but in the case of specific workloads, provides mediocre performance and may result in high energy and area overheads. On the other hand, application-specific integrated circuits (or accelerators) may be optimized for a specific algorithm, resulting in a high-specialized low-power architecture with few ad-hoc designed computing resources. Unlike general-purpose systems, application-specific devices are expected to perform a reduced spectrum of tasks within demanding area and energy constraints. Moreover, application-specific architectures hold great potential as an alternative mean to continue scaling application performance. This approach is well matched to embedded targets, which feature constant and well-known workloads due to narrow application domains.

Deep Learning and its power-hungry algorithms have placed a lot of emphasis on leveraging the characteristics of their workload to optimize inference. In this scenario, the need for architectures to address both computational and power demands increases. As a result, hardware accelerators play an essential role in implementing intense inference sessions on Application Specific Integrated Circuit (ASIC) scale

devices [61]. The software and hardware co-design techniques presented in the previous chapter are heavily used to exploit the features of Deep Neural Networks (DNNs).

Shifting the focus to the edge, it is clear how area and power limits play a crucial role. Here the optimization and acceleration of algorithms must be timely and judicious in order to respect all the constraints placed on area and power budgets. In this context, the design of ASIC-based accelerators for deep learning algorithms is tailored concerning specific workloads or workflows determined by careful algorithm profiling. Through a co-design work between software and hardware, we try to determine where the latter could play a fundamental role. For example, in the case of DL, the algorithms are composed of matrix-matrix or vector-matrix multiplications, so an accelerator is expected to provide considerable support in this direction. It is also possible to leverage some data flow properties or the data itself to further optimize execution. This chapter is divided into two macro sections: the first one discusses co-design with a more high-level (software) approach to training on edge, while the second one presents two hardware frameworks for accelerating convolution algorithms with leverage on data sparsity.

The elemental operation for fully-connected and convolutional layers is the multiply-and-accumulate (MAC). Since tens of thousands of MACs per layer occur, high performance requires developing a highly parallel structure able to allocate millions or billions of operations per second. Typically, the most used paradigms are two: temporal and spatial architectures, as explained in Chapter 2. Temporal ones are commonly represented by CPUs and GPUs, where a unique core control drives clusters of Arithmetic-Logic Units (ALUs). ALUs are decoupled, making communication or data transfer among them impossible. Indeed data only come from the memory hierarchy. On the other hand, hardware accelerators operate spatial architectures where ALUs, namely Processing Elements (PE), are distributed, forming a network where data can transit through local memory buffers. Since data fetch and relocation are two of the most costly tasks in terms of energy, architectures are tailored to ad-hoc data flows [61], in which data reuse trims off memory and energy consumption. For example, each MAC needs to fetch an activation and a weight for a total of two memory accesses, an additional memory access is required to write back the result. In order to overcome such bottlenecks, input and output stationary approaches have been developed. The former consists of reading the activations or weights once and then retaining these values through registers as

much as possible (according to the data flow). Data reuse is performed by shifting the information from PE to PE or giving all PEs access to such resources. On the contrary, the output stationary paradigm minimizes the energy required to write and read partial sums to and from memory. Partial results are retained and reused locally by allocating accumulators.

The objective of this chapter is to present a low-power architecture for edge computing capable of being integrated into a multicore platform such as PULPissimo.

The proposed design is both input and output stationary. Activations are shared among PEs and reused, while weights are continuously fetched. The peculiarity of this architecture is the ability to perform bit-serial operations. This reduces the required area and works efficiently with any bitwidth. The work is divided into two main parts: the development of a first basic architecture and its integration into PULPissimo, and a second part in which the same architecture is modified to take advantage of the sparsity of activations to improve performance.

4.1 Architecture

Accelerators are hardware units featuring high performance. Since they can perform just a few specialized tasks, they must be allocated in complex systems, working in synergy with a host microprocessor. As a consequence, each accelerator requires a suitable interface. In this current work, the accelerator, called SMAC-engine, is integrated into the PULPissimo platform as a Hardware Processing Engine (HWPE)[122]. PULPissmo is a parallel ultra-low-power microcontroller architecture designed to be embedded in an IoT endpoint [123]. Thus, the SMAC-engine would operate as a power-efficient extra-core for DNN edge processing. Its interface comprises a Tightly Coupled Data Memory (TCDM) and a microprogrammed Control Unit (CU). While the central core uses the first to transfer data to the accelerator, the latter orchestrates the elaboration phase. The memory bandwidth from the TCDM has been identified in 128 bits/cycle since this value provides a good trade-off between the area and frequency of the engine, as it will be reviewed in 4.1.3. The primary intent of PULP is to fulfill computation requirements imposed by IoT applications that demand continuous data stream processing capability at the cost of a few mW. In fact, since many devices such as cameras, microphones, accelerometers or drones

are battery-powered, overwhelming the pJ per operation boundary is becoming of extreme importance.

In order to take advantage of an accelerator in the PULP framework, data-flow, bitwidth, desired throughput and bandwidth must be carefully analyzed. About the data parallelism, the architecture aspires to present an adaptable solution able to process the state-of-the-art quantized neural networks found in the literature. Thanks to quantization techniques, the required bitwidths have been identified in 8 and 4 bits for the activation width, called Pa, and in 8, 6, and 4 bits for the weight width, called Pw. Such selection allows not to deteriorate classification tasks of the DNNs while simultaneously speeding up the computation and strongly reducing the memory occupation, especially for the weights, which overall number tends to be greater than the number of activations.

For the data flow concerns, an output and input stationary data flow is derived in which convolution is performed by fetching the activations along the channels' direction. Algorithm 1 describes the data flow loops required to complete a convolution task using the typical sliding window technique, where $\underline{x}$ is the input feature map, $\underline{y}$ the output one, and $\underline{w}$ the kernel. Six nested loops are needed to cycle first over the channel depth, then over the kernel size, up to the feature maps size. Moreover, two further loops serve for bit-wise operations. Shifting first the activations and then the weights from the Least Significant Bit (LSB) towards the Most Significant Bit (MSB), the engine produces partial results through bit-serial multiplications and sums. $n_W^{[i]}$, $n_H^{[i]}$, $n_C^{[i]}$ identify the width, height, and depth of the i-th feature map, while $f^{[i]}$ is the size of the i-th filter. 4.1 is a representation of the above-mentioned data flow where activations and weights are fetched along the channel direction to produce output feature maps in the same orientation. Such an approach allows us to retain activations for as many kernels as possible inside the considered convolutional volume and locally accumulate partial sums until the end of the task.

4.1.1 SMAC

The SMAC's basic structure, 4.2, is inspired by the one presented by Sharify et al. named Loom [124]. Loom architecture was inspired in turn by DaDianNao [87], where the design can perform M MAC operations per cycle concurrently. However, allocating M multipliers and an adder tree would severely affect the final

Algorithm 1 Convolution nested loops and bit-serial loops.

$ConvolutionLoops:$

1: **for** $h_{out} = 0 : n_H^{[i]}$ **do**
2: **for** $w_{out} = 0 : n_W^{[i]}$ **do**
3: **for** $k_{out} = 0 : n_F^{[i]}$ **do**
4: **for** $l = 0 : f^{[i]}$ **do**
5: **for** $j = 0 : f^{[i]}$ **do**
6: **for** $k_{in} = 0 : n_C^{[i-1]}$ **do**

$Bit - SerialLoops:$

7: **for** $wbit = LSB : MSB$ **do**
8: **for** $xbit = LSB : MSB$ **do**
9: $y[k_{out}][w_{out}][h_{out}] +=$
$x[k_{in}][w_{out}+l][h_{out}+j][xbit] \times W[k_{out}][k_{in}][l][j][wbit]$

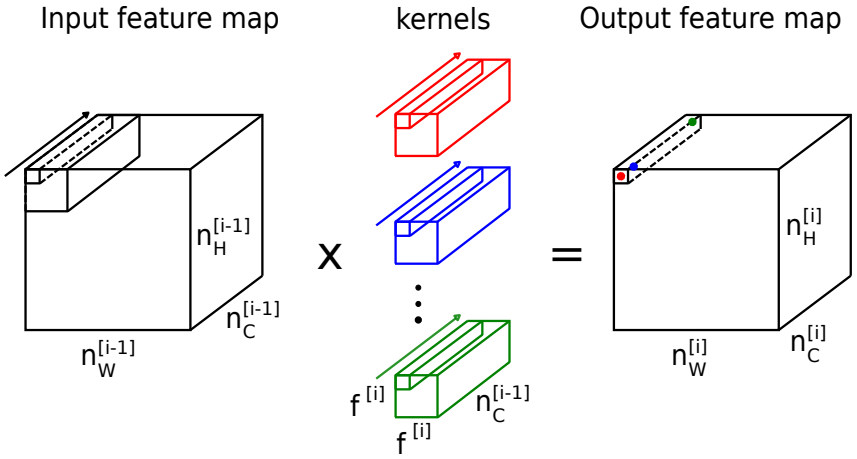


Fig. 4.1 Input/Output stationary data flow visualization. [12]

clock frequency, especially when M is large. AND gates can substitute multipliers transforming the parallel structure into a serial one, i.e. sequential multipliers. The throughput remains unchanged by increasing the number of processing elements. A number (M) of MAC operations equivalent to those performed by the parallel structure can be performed with the described structure. This configuration requires only the addition of two accumulators inserted after the AND block to retain the partial sums. Indeed, while the weight bits are stationary, the activation bits are shifted serially, and the partial sums relative to the fixed weight bit have to be summed together by a first accumulator AC1. The coherence among the partial sums is kept by shifting the content of AC1 to the right, in fact, every partial sum is

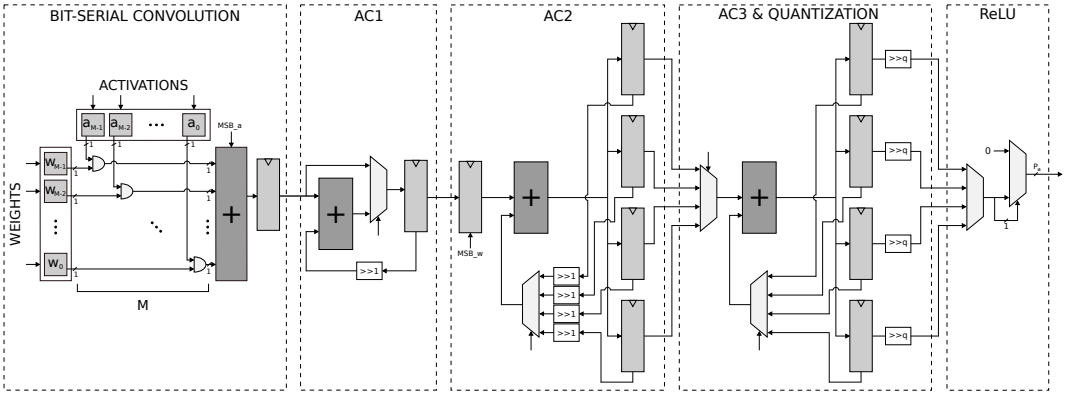


Fig. 4.2 Datapath architecture of a single SMAC block.[12]

produced with a different activation bit, i.e has a different weight. This is correct provided that activations and weights are shifted from the least significant bit (LSB) to the most significant bit (MSB), otherwise, the shifting would be performed in the opposite direction (left).

Following the same logic, a second accumulator AC2 is inserted after the previous one to accumulate the partial results obtained for each weight bit. Moreover, since data are represented in two's complements, a register between AC1 and AC2 allows inverting the content of AC1 whenever the MSB weight bit is selected. Unlike AC1, AC2 is composed of 4 registers. This allows retaining the partial sums of four different filters in parallel. A third and final accumulator, AC3, is inserted to sum together partial results of the same convolutional volume. In order to keep consistency with the previous stage, four registers are also used in AC3. The bitwidth of AC3 concerns the dimension of the most extensive layer to be processed in the target DNN. Thus, a parameter called the Maximum Number of Operations (MNO) is derived. Since our architecture processes M activations per time, MNO depends on the maximum number of $1\times1\times$M volumes present in a convolutional layer. In the case of VGG16, MNO would be $3\times3\times512/M$, where 512 is the largest channel dimension with 3×3 kernels.

After the convolution operation is completed, the result must be quantized to be represented with the same bitwidth of the input activations, in this case, Pa bits. The amount of shifting to be performed on the result strictly depends on the size of the convolutional layer. However, instantiating barrel shifters to accomplish quantization is not highly convenient, primarily due to the area overhead, which would lead to an unacceptable area occupation. A simple but effective solution is

to perform quantization serially using shift registers in the AC3 stage. Hence, once the convolution operation is done, a programmable counter can be exploited to shift the registers' values serially. Finally, one of the most used activation functions, the Rectified Linear Unit (ReLU), has been inserted after the quantization step. The ReLU is performed thanks to a multiplexer that uses the MSB of the convolution output as input selection.

4.1.2 SMAC-engine

The bitwidth of the hardware datapath has been designed in the worst case, i.e., considering the maximum bitwidth both for activations and weights, Pa = 8 and Pw = 8. Unlike the solution with parallel multipliers, the serial one requires increased latency to perform M MACs. Instead of M MAC/cycle, it can perform M MACs after Pa $\times$Pw cycles. Therefore, for the serial solution to match the same throughput as the parallel one, the number of MAC blocks must be augmented by Pa$\times$Pw times. Nevertheless, the previous assumption is still reasonable because the serial solution employs far less hardware than the parallel counterpart, thus the required area is contained.

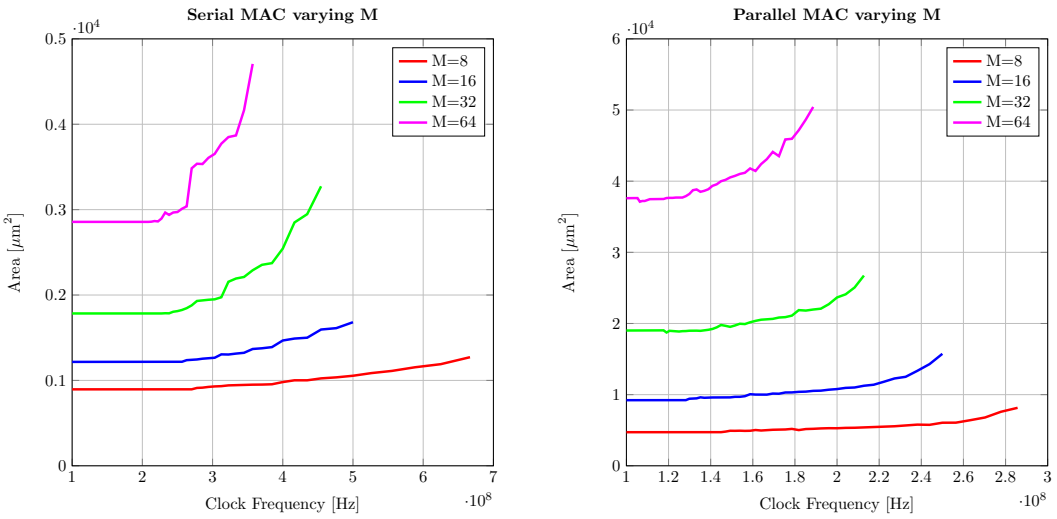


Fig. 4.3 Serial vs. paralllel approach for a single SMAC block.

In order to establish an acceptable trade-off for M, a comparison in terms of area vs. operating frequency between parallel and serial approaches has been performed. Using the UMC-65 nm library in worst-case conditions, namely 0.9 V supply voltage

@ 125 °C, we found that even if the SMAC block is replicated Pa×Pw times, the area for the parallel case is still 2.91× larger, as showed in Figure 4.3. Indeed, as mentioned above, a SMAC block demands far less area than the parallel counterpart (around an order of magnitude less) and can reach almost twice the maximum frequency. This analysis has demonstrated that a reasonable trade-off among area, memory bandwidth, and operating frequency is obtained with M = 16, as described in 4.1.3.

In conclusion, the SMAC-engine is a cluster of Pa×Pw = 64 SMAC blocks conceived, as mentioned, to support the worst-case (Pa = 8 and Pw = 8) and with M = 16. Activations are shared among SMACs, so the engine can process M activations per convolution, while each SMAC can take care of M weights per kernel. Thus, the SMAC-engine can process 64 kernels per convolution, which can be extended to 256 thanks to the four extra registers in AC2 and AC3 stages. This denotes the upper boundary related to the number of filters. When more than 256 kernels are involved, the structure handles them by dividing the process into two parts, each working with 256 kernels and fetching the same activation twice.

Besides the datapath, the SMAC-engine is supplied with a low-level Control Unit (CU) in charge of orchestrating the convolution operation. It can handle kernels up to 3×3, but potentially it can be expanded to operate with larger filters, provided that the accumulator AC3 is sufficiently sized to store additional partial results. Low-level CU comprises two main modules: a 16 states Finite State Machine (FSM) able to drive the DP and a cluster of counters required to organize the FSM evolution. In addition, the FSM generates status signals demanded by a high-level FSM that manages address generation and the memory interface as described in 4.1.3. Counters are programmed thanks to a latch-based register file placed in the high-level control that is written by the processor present in PULPissimo [123]. At the beginning of the convolution operation, while activations and weights are uploaded to the engine, counters are updated with the corresponding value computed based on the current convolutional layer size.

4.1.3 PULPissimo-integration

One of this work's main contributions is to deliver a hardware accelerator fully integrated into a microcontroller-based platform that enables DL in an IoT node. The

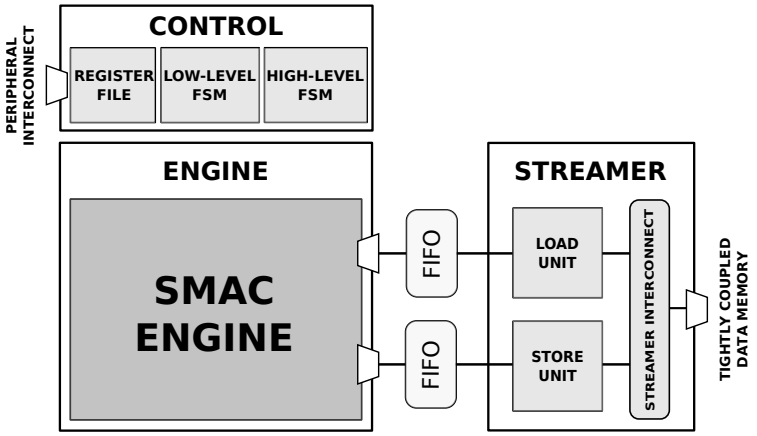


Fig. 4.4 HWPE architecture. The SMAC-engine is integrate in PULPissimo by means of a control and a Streamer. The first orchestrates the elaboration thanks to a Low-Level and a High-Level FSM and a register file, while the second manages the data streams.[12]

target platform is PULPissimo since it offers the opportunity to integrate accelerators in the form of HWPE, and it is entirely open-source. There is no straightforward communication between the host processor in PULPissimo and the HWPE, but data (activations and weights) are written in the TCDM by the former and fetched by the latter. Since the TCDM is shared with other modules present in the SoC, a mechanism that manages the accesses is required, as described in [125]. Such an element is called Streamer. It follows a handshake protocol to establish a directional link through which data streams can flow. Whenever different clock domains are present, a FIFO can be used to connect a Load Unit and a Store Unit, avoiding losing any information. Excluding the Streamer, another interface is essential to program the accelerator by loading the counters' values accordingly to the size of the convolutional layer to be processed. This is the reason HWPE presents a peripheral connected with a memory-mapped register file, as illustrated in 4.4. The host core present in PULPissimo exploits such peripheral to upload setting values for the counters (present in the low-level CU) and the microprogrammed high-level CU.

As previously noted, the maximum bandwidth is 128 bit/cycle. This bandwidth determines how the Streamer handles the TCDM both during reading and writing operations. Considering the maximum activations and weights bitwidth, it is possible to fetch either 128/Pa = 16 activations or weights per cycle. So this is another reason to choose M = 16. Considering M = 16, the parallel implementation (with such available bandwidth) would allow performing 16 MAC/cycle. In two cycles, it is possible to fetch 16 activations and 16 weights working with Pa = 8 and Pw

= 8. Equivalently, to obtain the same results with the SMAC blocks, one would need to replicate the SMAC structure Pa×Pw = 64 times. Again, the hypothesis of having a SMAC-engine composed of 64 SMACs is confirmed. Since the data flow is thought to be input stationary, the structure is replicated in a direction that maximizes the usable kernel number with the same activations (activation reuse). Using the available bandwidth to fetch 128 weight bits per cycle (instead of 128/Pw = 16 weights on the entire bitwidth), in Pa = 8 cycles (required to shift the activation bits), it is feasible to achieve a total of 128×8 = 1024 bits, that is precisely the number of bits necessary to feed 64 SMAC blocks, each working with M = 16.

4.1.4 Performance and considerations

The Serial-MAC-engine has been synthesized with a 65 nm UMC technology in worst-case conditions (0.9 V @125 °C) leading to 0.36 mm^2 silicon area coverage. VGG16 and SqueezeNet have been used as testbench implementing different parallelism widths to validate the architecture and define its behavior.

Table 4.1 Comparative table.[12]

HW solution		Technology [nm]	Frequency [MHz]	Area [mm^2]	Throughput [GMAC/s]	Power (@ 25°C) [mW]	Energy Efficiency [pJ/MAC]	Throughput/Area [GMAC/s/mm^2]
SW	CMSIS-NN [126]	-	-	-	0.05 - 0.15	1 - 1000	667 - 2000	-
	PULP-NN [127]	-	-	-	0.5 - 1	1 - 100	40 - 67	-
bit-parallel	ShiDianNao [8]	65	1000	4.86	64	320	5	13.17
	Eyeriss [85] @ 1 V	65	200	12.25	23	278	12.09	1.88
	BRein [128] @ 1 V	65	400	3.9	518	600	2	132.82
bit-serial	UNPU [129] @ 1.1 V	65	200	16	827	5.92	0.36	51.69
	Stripes [130]	-	-	16	345.6	297	0.86	21.6
	Loom [124]	-	-	-	-	-	-	-
	SMAC-E @ 1.0 V *	**65**	**476**	**0.36**	**14.28**	**7.38**	**0.58**	**40.8**
	SMAC-E @ 1.2 V *	**65**	**625**	**0.36**	**18.75**	**12.91**	**0.76**	**52.08**

* SMAC-E results refer to Pa=8 and Pw=4

4.1 compares the SMAC-engine with other state-of-the-art architectures dedicated to DNN algorithms processing. Considering that a MAC in convolutional networks represents the essential operation, we used the energy efficiency metric, defined as the energy needed to perform a complete MAC. The SMAC-engine with its 0.58 pJ/MAC can be compared with the efficiency required by UNPU [129]. Even if the latter has a higher throughput, our implementation features a 44.4× smaller area, crucial in area-constrained platforms such as IoT edge nodes. Moreover, with the same silicon coverage, the performance would be almost equivalent, as portrayed by the throughput to area ratio in the last column, Table 4.1. Furthermore, while UNPU

is a stand-alone accelerator, our work is already integrated into a microcontroller-based system and ready to use. The SMAC-engine outperforms Eyeriss [85] and BRein [128] in terms of energy efficiency while being comparable to Stripes [130] that follows the same serial approach. Although the architecture was inspired in part by Loom [124], unfortunately, the latter does not offer an implementation on any technological node, making the comparison incomplete. The SMAC-engine, thus, represents a possible detailed implementation of a Loom alike architecture.

Software strategies like [126] and [127] offer high-level flexibility suitable for several different hardware platforms, boosting their applications. However, our ASIC-based approach represents a DNN-tailored hardware accelerator that optimizes every step from data transaction to the MAC execution.

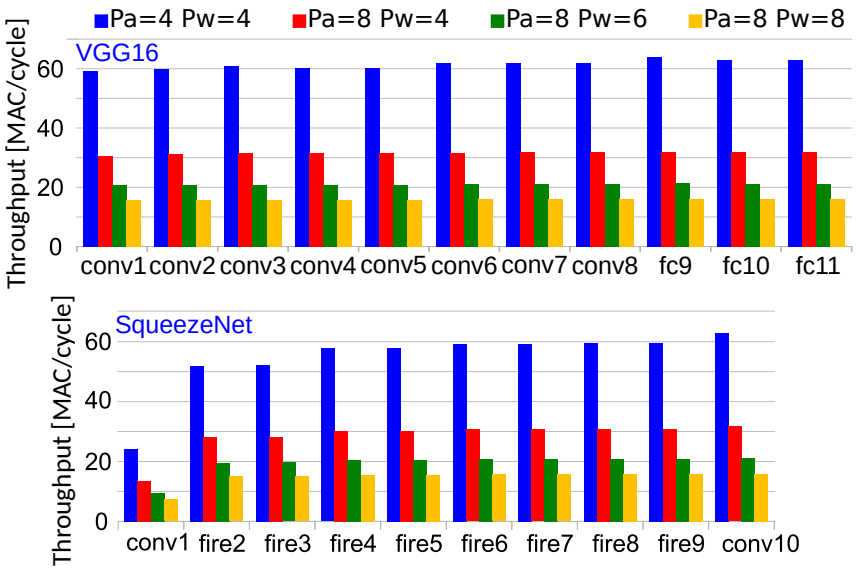


Fig. 4.5 Layers throughput over several bitwidth combinations.[12]

The current design represents an instance of how the SMAC-engine could scale. We fixed M = 16 and the maximum width of activations and weights to 8 bits. However, potentially the architecture can be enlarged to accommodate more general cases or tailored to a specific instance where the optimization has to be pushed to the limit. Even the maximum kernel size can be adjusted by revising the AC3 width. While the DP bitwidth needs to be tuned according to the application to be tackled, the CU remains unchanged since it can be programmed through the external peripheral. Additionally, as SMAC blocks are replicated to share activations, they

can be replicated to share weights in a systolic array fashion (this will be investigated as future work).

Figure 4.5 illustrates how in VGG16 and SqueezeNet, the throughput, intended as the number of MAC per clock cycle, remains unchanged over the kernel sizes according to different values of Pa and Pw. Therefore, the bottleneck is defined by the data parallelism regardless of which layer is processed. Therefore, as mentioned above, the SMAC-engine parameters can easily be scaled so that the designer can tune features to meet the required throughput. The efficiency drop in Squeezenet's conv1 is due to the fact that the 7x7 kernel has been decomposed into two 3x3 kernels that slow down the execution.

Table 4.2 Number of cycle and power consumption with different data bitwidth related to a filter of size $3 \times 3 \times 128 \times 128$.[12]

Bitwidth	# cycle [M]	Throughput [MAC/cycle]		Power [mW]
		SMAC	SMAC-engine	
Pa=4, Pw=4	30.84	0.937	59.97	7.38
Pa=8, Pw=4	58.92	0.490	31.39	7.38
Pa=8, Pw=6	88.67	0.326	20.86	7.41
Pa=8, Pw=8	117.57	0.246	15.73	7.42

Results obtained @ 1.0 V with an operating frequency of 476 MHz

Table 4.2 shows how data bitwidth affects the latency and the power consumption having as target the third layer of VGG16. This circumstance stresses the SMAC-engine to operate with a number of filters greater than the default one (64), resulting in one of the most power-hungry configurations. The number of cycles refers to the clock latency required to load activations and weights, process the total convolutional volume, and store the results back into the memory. Since multiplications are performed serially, the throughput decreases by expanding data width, as depicted in the third and fourth columns, where both single SMAC and SMAC-engine (64 SMACs) are reported. Even though the required clock cycles increase 4x going from the narrower bitwidth to the larger one, the power undergoes a negligible increase of just 40 µW, thanks again to the bit-serial paradigm. This is an outstanding indicator that hardware usage is constant, and there are no parts that waste energy when working with reduced bitwidth.

4.2 Enhancing Performance with Sparsity

One possibility to improve the architecture's performance is to take advantage of data sparsity. Generally speaking, CNNs are characterized by an elevated sparsity level on both weights and activations. As described in Chapter 2 Section 2.4, 2016 has been a productive year, with multiple sparsity-aware accelerator designs presented.

Sparsity affects both weights and activations. In the former, sparsity results from quantization and pruning processes, which constrain several values to zero while trying to reduce the complexity of the data. On the other hand, the sparsity of the activations comes mainly from the ReLU activation function extensively used in CNNs. This source of sparsity tends to have a more significant impact than the former and increases in deeper layers of the network.

In the following implementation, the goal is to improve the performance of the SMAC-engine architecture by exploiting the sparsity of activations. Unfortunately, the design requires more complicated control logic and local memory, introducing power and area overhead. In addition, one of the significant challenges of exploiting sparsity is that convolution goes from being a dense, structured operation to an unstructured operation. Avoiding operations involving null elements means no longer having deterministic computation patterns but determining which operations to accomplish and which to skip on the fly.

This section will first present a case study of activation sparsity and its distribution within a CNN. Then the new sparse SMAC engine is described and compared to the base implementation. In the end, the results and conclusion are discussed.

4.2.1 Activation Sparsity Distribution Analysis

This section analyses the activation sparsity behavior in a CNN. The goal of such an analysis is to infer the distribution of the sparsity of activations with respect to the depth of the network, i.e., a layer-wise distribution, and with respect to the direction of the channels rather than the width or height of the feature maps. The first distribution is needed to profile the percentage of null activations in a network and the variation with respect to its layers. The second distribution helps to understand whether sparsity varies in the three spatial dimensions of feature maps and which data stream might benefit most.

For this analysis, the network model considered is the current state-of-the-art CNN, namely ResNet [4]. In particular, the analysis comprehends two models of different sizes: ResNet 18 and ResNet 34. The intent is to see if the distributions remain consistent as depth changes.

The Tiny Imagnet [131] is used as a reference dataset for inference, a smaller version containing 200 classes over 1000 present in the well-known Imagenet.

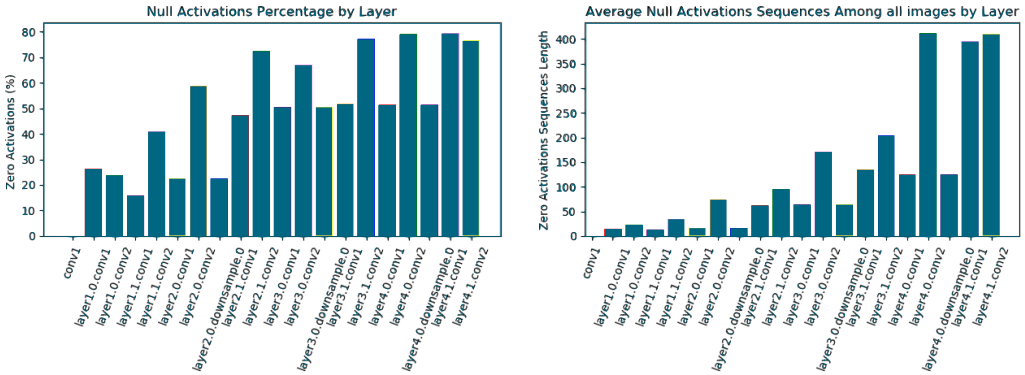


Fig. 4.6 Sparsity Analysis for ResNet 18 (left), average zero sequences in channel direction (right).

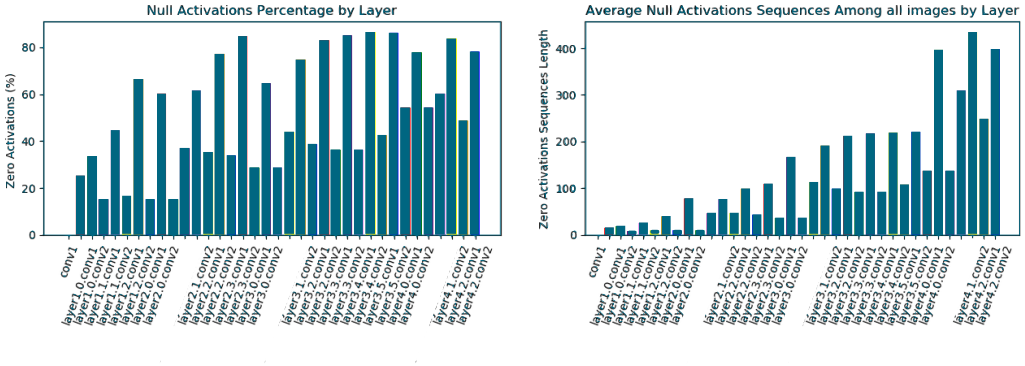


Fig. 4.7 Sparsity Analysis for ResNet 34 (left), average zero sequences in channel direction (right).

Figures 4.6 4.7(left) show how the activations sparsity increases with channel depth. Sparsity ranges from 20 to 80% in both NNs, with an average of around 50%.

This distribution is noteworthy and describes how by going deeper into the network, the number of valuable operations is conspicuously reduced; moreover, on average, 50% of convolutional operations could be avoided, significantly speeding

up operations. Note that the first layer has no null elements; in fact, the first layer typically represents the input image, while subsequent layers are affected by the ReLU.

This analysis was also repeated for other networks, such as VGG16, showing similar behavior.

As for the sparsity distribution in the three dimensions of the feature maps, the sparsity is equally distributed. We have not noticed determining patterns that made us think one dimension is more affected than others. To better complete the analysis, two graphs present the relation between the activation sparsity and the channel direction (direction in which activations are fetched in the SMAC-engine dataflow).

From Figures 4.6 4.7(right), it is possible to observe the average of the sequences of null elements in the direction of the channels per layer. Here, it can be seen how the sequences gradually become longer with the depth of the network. These metrics are helpful in understanding which data compression technique might be most effective in reducing memory requirements.

4.2.2 Data Compression and Dataflow

As reported above, the percentage of sparsity in CNNs appears to be considerable, with an average of 50% per layer. In this regard, in order to reduce the space occupied in memory by the activations, it is possible to store the scattered feature maps in a compressed format. This compression consists of saving only the non-zero activation values and indices, needed to reconstruct their position. Several compression methods are reported in the literature as illustrated in Chapter 2 Section 2.4.

For this work, the sparsity map method is chosen, i.e., in addition to the vector of nonzero elements, an index matrix with as many elements as the feature map indicates where the values are nonzero (logical 1) and where they are null (logical 0). The reason for this choice lies in the excellent compression ratio and the quality of being hardware-friendly. In fact, the sparsity map requires only one bit per matrix element and allows the input feature map to be easily reconstructed. The compressed format has the advantage of reducing the memory requirements. However, it introduces complexity into the architecture since it will require a decoder to compute the relative indices of the weights to be convolved with the respective activations and an output

encoder that produces a sparsity map along with the nonzero values. Avoiding non-zero operations does not come for free, but some overhead in terms of area and power will be needed to ensure that data, indexes and decoding/encoding algorithms are congruent since the convolution is now unstructured.

Unlike the SMAC engine base implementation, the data will be compressed in both input and output in the sparse implementation. The compression, however, represents one source of overhead in power and area since it will require extra logic for the decoding/encoding phase.

As mentioned, the sparsity map is hardware friendly, and through its indexes, it is possible to compute the addresses of the respective weights to fetch. This last operation is critical for working with sparsity since avoiding null operations involves accessing the weights in a noncontinuous manner in memory. This is impossible via TCDM since the memory bandwidth is fixed and does not allow a single-weight read. Consequently, many of the weights read from memory would then be unused, wasting much of the energy expended in the process.

The only way to have such flexibility is to accommodate the weights in on-chip memories. Thus compared to the previous implementation, kernels will be preemptively loaded into local memories that will be consulted whenever new non-zero activations are loaded. However, this again introduces an overhead in area, energy, and since much data has to be moved, in time, i.e., the load cycles will slow down the execution of the CNN layer. So to allocate the kernels, local 512B single port memories will be prepared. Each memory will contain one channel of a 3x3 kernel, consequently, each kernel requires 9 memories. The number of memories and their arrangement will be discussed later.

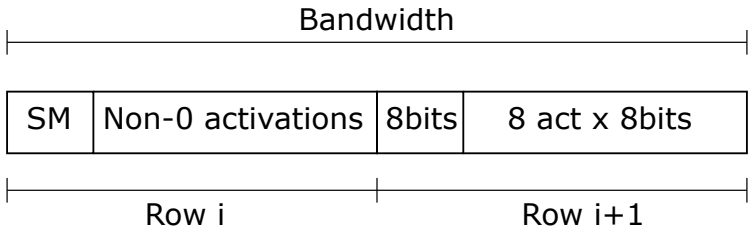


Fig. 4.8 Available bandwidth towards TCDM and activation distribution fetch.

Compared with the base implementation, the bandwidth to the TCDM increases from 128bit/cycle to 144bit/cycle (note that this bandwidth is programmable on Pulpissimo).

There are two main reasons for a larger bandwidth: first, the weights have to be preloaded into the local memories; second, the new data flow requires access to more activations channels at a time. Moreover, 144 is not only divisible by 4, 6 and 8 but also allows loading 8 activations and 8 sparsity indexes from two rows in parallel, as shown in Figure 4.8.

As mentioned, the data flow undergoes a slight change. In fact, the activations will still be fetched in the direction of the channels as before, but the new structure reads up to 4 rows of the input feature map in parallel, producing two outputs (two output feature map rows) at a time. This increase in input activations flux allows massive reuse of kernels saved in local memories. The data flow is shown in Figure 4.9.

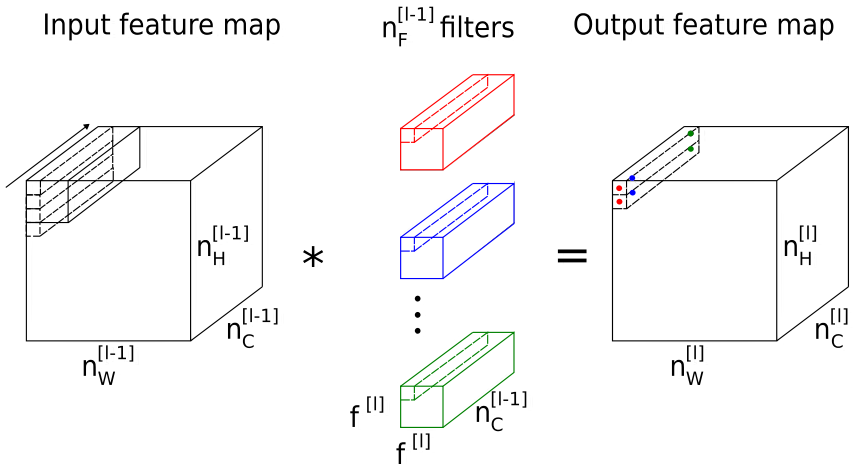


Fig. 4.9 Dataflow for sparse SMAC-engine.

Activations will continue to be shared along rows, but there will be up to 4 columns of SMAC, one for each row of the input feature map read. In addition, kernels will be shared along columns. The kernels will be divided into three portions compared to the base implementation. Considering a 3x3 kernel, this will be divided into three rows, each containing three kernel channels, which will occupy as many rows of SMAC as in the figure. This allows for maximum kernel reuse and up to 3 sliding windows to be computed simultaneously. Further details on the architecture will be provided in the following section.

4.2.3 Sparse SMAC engine

In this section, implementation details regarding data parallelism will no longer be discussed since the sizing procedures are the same discussed for the SMAC-engine, but only details regarding the architectural composition. The activations can be on 8/4 bits and the weights on 8/6/4 bits, and everything has been sized considering the worst case, i.e., Pa and Pw equal to 8.

The first significant change w.r.t. the base SMAC architecture is related to the data flow. In fact, as mentioned before, in order to take advantage of sparsity to speed up convolution, it is necessary to load weights into local memories.In addition, since memory reads are energy-consuming, it is necessary to reuse the same weight as much as possible for a low-power approach. Consequently, the input flow of activations must also increase to ensure good reusability of the weights. The new SMAC-block structure consists of 6 SMACs and 9 memories capable of holding a kernel, as shown in Figure 4.10.

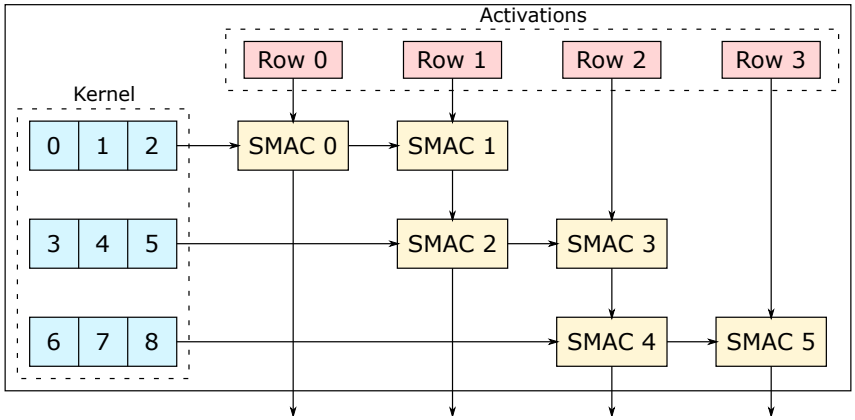


Fig. 4.10 Sparse SMAC-block.

Kernel values are shared along columns, while activations are shared along rows (up to 4 rows of feature map input at a time). This scheme makes it possible to reuse weights and activations as much as possible. In addition, note that activations are shared along the entire length of the columns, thus also with other SMAC-blocks. In fact, multiple SMAC-blocks can be used in parallel by replicating this structure vertically. Each of the SMAC-blocks works with a different kernel contributing to a different channel of the output feature map. This approach allows the reuse of the addresses of the weights. In fact, since the data are compressed, it is necessary to compute the address of the weights corresponding to non-zero activations. This

operation requires an energy overhead, however since the activations are shared among all SMAC-blocks, it is possible to reuse these indexes.

The current SMAC-block configuration allows the processing of up to 4 activation rows simultaneously, producing two output feature map elements at a time. Moreover, each column shares an entire kernel row, i.e., three adjacent channels, as shown in Figure 4.10. This scheme allows computing three sliding windows (stride equals 1) concurrently. In fact, SMAC 0 can convolve the same set of activations with channels 0, 1, and 2 of the kernel, just like SMAC 1. This again allows for a more comprehensive reuse of weights and activations, although it requires slightly more complex control. Not all kernel channels need to be multiplied by all activations, e.g., kernel channel two will never be multiplied by the first two columns of the input feature map in the case of no padding. The low-level control unit provides synchronization signals for these automatons, which will be different and more complex than the base implementation.

A further consideration is required about the low-level control unit. Neural Networks have sparsity statistics such as those described in the Subsection 4.2.1. The average sparsity tends to increase with the depth of the network, however, the sparsity is not always equally distributed in the feature map; there will be areas that are more sparse than others.

Taking advantage of a data flow that reads in the direction of the channels tends to mitigate this problem of unbalanced distribution; although some feature maps are more sparse than others, reading in the channel direction, the average sparsity may remain constant.

However, it is impossible to guarantee equally distributed sparsity, consequently, different rows will have different numbers of nonzero elements when loading activations. This results in different processing times, especially near the end of the channel, where nonzero elements fail to fill the SMAC matrix completely.

Therefore, the low-level control unit is divided into four autonomous units, each one controlling a column. In this way, each SMAC belonging to the same column will be decoupled from the adjacent one. Finally, of course, a synchronization mechanism must be provided for the summation of partial results. Each SMAC block can process an entire kernel, consequently, to be able to work on several kernels at the same time, this structure is repeated 16 times in the vertical direction (shared activations).

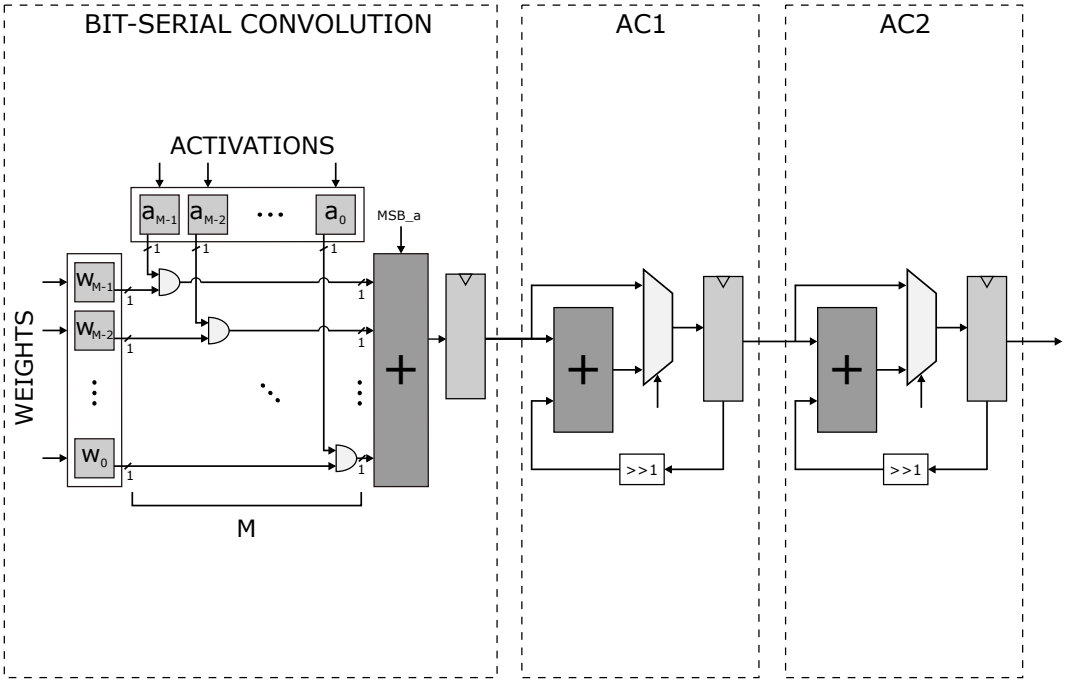


Fig. 4.11 SMAC architecture for sparse convolution.

The architecture of a single SMAC is depicted in Figure 4.11. Just as in the basic implementation, M is set equal to 16, so each SMAC can process 16 activations at a time. After the matrix, two stages of accumulation are needed since the multiplication operation is bit-serial. The first AC1 accumulates partial results from the shift of activations, while the second AC2 handles with weight shift. Compared with the basic architecture AC2 has only one accumulation register and no longer four. Another substantial difference lies in AC3, in fact, this accumulator is no longer part of the single SMAC but is shared among groups of 3 SMACs, as in Figure 4.12. In particular, considering Figure 4.10, SMACs 0, 2, and 4 will share an AC3, as well as SMACs 1, 3, and 5. With the new data flow the kernel is divided over three rows, consequently AC3 must work in a shared way between the rows for accumulating elements belonging to the same convolutional volume. AC3 goes from 4 to 3 accumulation registers. Each contains the partial results of a sliding window (stride = 1).

AC3 performs the ReLU on each output produced. After the activation function, a compressor is needed to save the output feature map in a compressed way. To do this, two data streams will be needed, one related to the sparsity map and one related

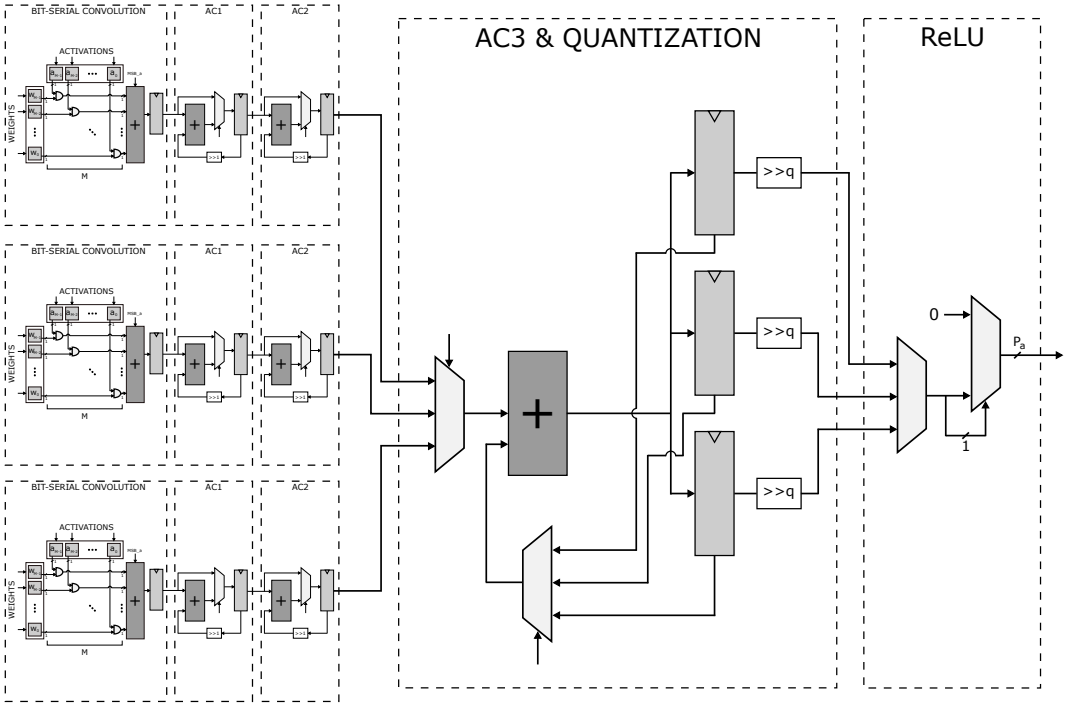


Fig. 4.12 Third accumulation stage in sparse architecture.

to non-zero activations. Whenever the 16 smack blocks produce a result, they are compressed and written to the TCDM.

4.2.4 Performance and considerations

The Sparse Serial-MAC-engine has been synthesized with a 65 nm UMC technology in worst-case conditions (0.9 V @125 °C) leading to 1.8 mm^2 silicon area coverage. Concerning the base implementation, the area increased by 5 times. This is mainly due to 2 factors. The first factor concerns the area overhead due to the sparsity control logic that includes reading the sparsity map and generating memory indices that contributes to a 10% increase in area. Second, the largest contribution undoubtedly comes from local weight memories that were not present before. Using 9 local 512B memories per SMAC block results in a total of 144 memories representing 1.2 mm2, or 66% of the entire design.

The first factor concerns the structure of SMACs. The previous design was based on 64 SMACs, while each of the 16 sparse SMAC blocks is composed of 6 SMACs for a total of 96 SMACs. Although this type of SMAC is smaller since AC3 is shared

and AC2 has only one accumulation register, the total number of macs leads to a slight increase in area.

The previous design was based on 64 SMACs, while each of the 16 sparse SMAC blocks is composed of 6 SMACs for a total of 96 SMACs. Despite the 1.5x increase in the number of SMACs, the computational core area remains almost unchanged, in fact the type of SMACs used now is more compact than the previous one. In particular, AC3 is shared in groups of 3 SMACs, while AC2 has only one accumulation register.

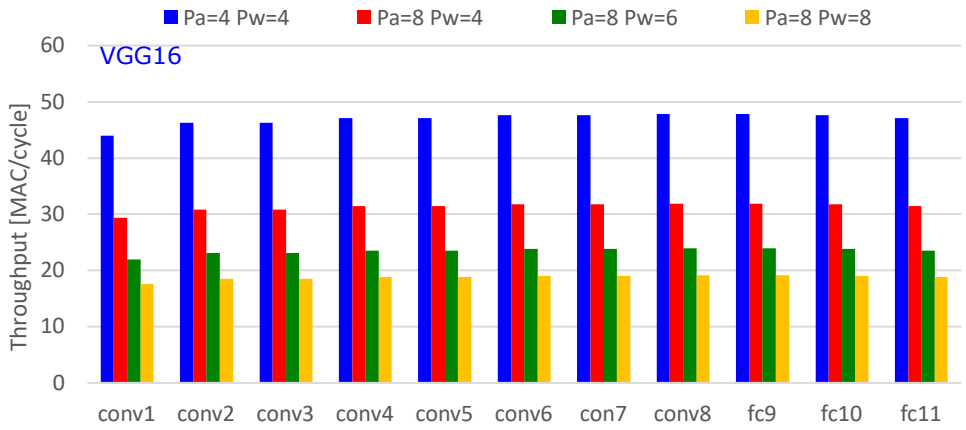


Fig. 4.13 VGG16 throughput w.r.t. all the bitwidth combinations with sparsity 0%.

In order to have a fair comparison with the previous structure, VGG16 was again used as a benchmark. Figure 4.13 illustrates how in VGG16 (sparsity 0%) the throughput, understood as the number of MACs per clock cycle, remains unchanged for the kernel size under different values of Pa and Pw. Again, the bottleneck is defined by data parallelism, regardless of which layer is processed. The Sparse SMAC engine, with dense VGG16, achieves peak throughput for Pa=4 and Pw=4 of about 48 MAC/cycle, lower than the 63 MAC/cycle of the previous implementation despite the 1.5x increase in the number of SMACs. The reason is that the number of MACs per cycle also takes into account the loading cycles of the weights and the overhead cycles required to handle sparsity and compute the memory addresses from which to extract the correct weights. This contributes to reducing the average throughput over the entire layer.

Considering the Conv8 layer of VGG16 using a 3x3x128x128 kernel, Figure 4.14 analyzes how thorughput changes as the percentage of sparsity changes. In fact, by exploiting sparsity, this architecture can avoid operations involving null elements,

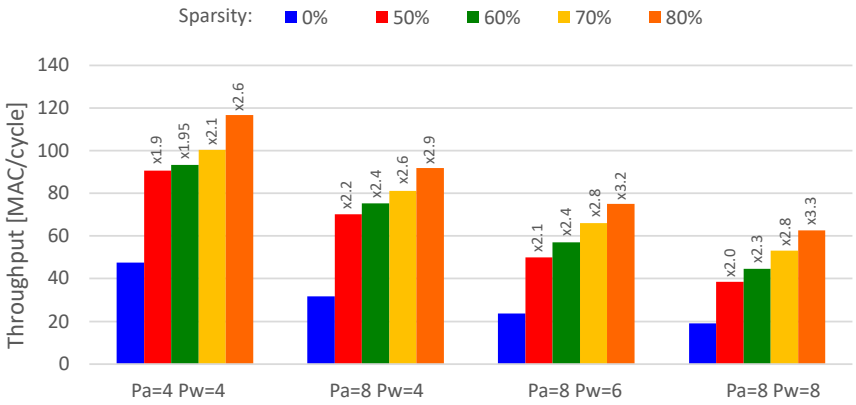


Fig. 4.14 VGG16 Conv8 sparsity analysis.

virtually increasing its throughput. By skipping some of the operations, the number of total cycles per layer can be drastically reduced. This effect is further amplified by the fact that multiplications are serial. Proportionally, the increase in throughput occurs gradually and with very similar ratios for all combinations of parallelism as sparsity increases.

The results shown in Figure 4.14 are significant. They confirm that the possibility of exploiting sparsity to optimize CNN execution is in fact possible. Already at 50% of sparsity, which, as mentioned at the beginning of the section, is an average and widespread value in CNNs, the throughput undergoes a 2x increase, up to 3x average for sparsity of 80%, with a peak of 3.3x for the Pa=8 and Pw=8 case. The absolute throughput values reach 116 MAC/cycle peaks in the Pa=4 Pw=4 case, with an increase of more than 1.8x compared to the base implementation's maximum throughput.

As mentioned earlier, data parallelism represents a bottleneck against MAC/cycles. This graph emphasizes this further, showing how throughput decreases progressively from Pa=4 and Pw=4 to Pa=8 and Pw=8.

Table 4.3 Comparison between base implementation and sparse implementation for Pa=8 and Pw=4 and sparsity 0%

HW solution	Technology [nm]	Frequency [MHz]	Area [mm^2]	Throughput [GMAC/s]	Power (@25°C) [mW]	Energy Efficiency [pJ/MAC]	Throughput/area [GMAC/s/mm^2]
SMAC-E @ 1.0V	65	476	0.36	14.28	7.38	0.58	40.8
SMAC-E @ 1.2V	65	625	0.36	18.75	12.91	0.76	52.08
SSMAC-E @ 1.0V	65	476	1.8	15.16	36.9	2.43	8.4
SSMAC-E @ 1.2V	65	625	1.8	19.91	64.55	3.24	11.06

The table 4.3 compares the base implementation (SMAC-E) and the sparse implementation (SSMAC-E). The results are shown for Pa=8 and Pw=4, considering a sparsity of 0%. As seen in the Sparse SMAC engine, the area is about 5 times higher than the previous one. The memories in fact contribute to an increase in area and power. However, it must be said that while the throughput and energy efficiency are fixed for the base implementation, this can vary in the case of the sparse implementation. In fact, considering the same data parallelism, with sparsity at 50% we would have a de 2.2x increase in throughput and a power efficiency of 1.1 pJ/MAC, up to 0.83 pJ/MAC with 80% sparsity, comparable with base implementation.

The current design represents an example of how the sparse SMAC engine might scale. We set M = 16 and the maximum width of activations and weights at 8 bits. However, potentially the architecture can be extended to meet more general cases or adapted to a specific instance where optimization needs to be pushed to the limit. Compared to the basic implementation, the sparse architecture has a few more limitations. For example, every time a SMAC block is added, it must be considered that in addition to 6 SMACs, 9 memories are also added. This has a non-negligible impact on the average area and power. In addition, expanding the number of SMACs in each block, for example, by expanding the number of columns, would require reading more activations and more complex management of the weight address production mechanism.

4.2.5 Conclusions

This chapter presents the Serial-MAC-engine (SMAC-engine), a 65 nm convolutional hardware accelerator with variable parallelism suitable for quantized neural networks. The 14.28 GMAC/s at the cost of 0.58 pJ/MAC @ 1.0 V have been achieved exploiting a serial approach where multiplication is performed bit by bit, reducing enormously the hardware required for multipliers and consequently boosting the maximum accessible frequency. The wide choice in the data parallelism enables the architecture to make inference on several applications and networks targeting the Internet-of-Things paradigm.

The architecture is then further developed to take advantage of the sparsity of activations to improve performance. However, the Sparse MAC engine requires a new data flow and a revised structure, introducing significant overhead in terms of

area and energy. However, the throughput results improved almost by a factor of 2x over the previous implementation reaching peaks of 56.88 GMAC/s for Pa=8 and Pw=4 and sparsity 80%.

The accelerator is fully integrated into a microcontroller-based platform, namely PULPissimo.

Chapter 5

Functional Verification beyond ISA architectures.

Verification guarantees that the design specifications are met and has become an essential aspect of the product development cycle of digital hardware. Functional verification has become more and more critical as the complexity of the hardware grows, requiring costly and time-consuming efforts, up to 60% of the product development cycle. Functional and logic flaws account for 75% of causes for a redesign of more than two-thirds of ASIC [132]. Of this 75%, more than 80% consists of design errors, while the remaining are attributable to incomplete specification or internal and external IPs. The market requires scalable verification options and revolutionary metamorphoses in design methodology to overcome the problem of verifying complex designs. Increasingly, the conventional black-box verification procedures are substituted by white-box verification methodologies. Here, verification engineers are involved in the early design phase in order to have an architecture prone to verification.

This chapter is dedicated to the verification framework. The structure is the following:

- beyond ISA architectures, challenges, and future trends: explains the nowadays verification challenges and how the market is evolving;

- The basic structure of a simulation UVM-based testbench: introduces how the functional verification is performed and how the architecture presented in Chapter 4 has been verified;
- Conclusions.

The main contribution of this chapter is to complete the workflow and give context to an often underestimated but increasingly important topic in modern industry. The narrative began with codesign for software, then moved to hardware, and concluded with verification.

5.1 Beyond ISA Architectures, Challenges, and Future Trends

Nowadays, hardware verification is a reasonably mature subject, both in industrial and research practice. In industrial exercise, verification is currently affirmed as a vital element of the system development flow. Unfortunately, despite decades of advancements, a significant gap still exists between state of the art in technology and the verification requirements for modern complex designs. The rapidly transforming design ecosystem exacerbates the status as the market is rapidly moving towards the era of the Internet of Things (IoT), automated vehicles, etc.

This era creates an environment where every electronic device can gather, analyze, and store sensitive information, such as location, health status, and documents. Such information can then be communicated to billions of other computing devices that operate without pause, forming a complex network that may incorporate millions of potentially malicious nodes. One of the most critical consequences of IoT on verification is the number of available resources. With the increasing demand to produce billions of computing devices, time-to-market conditions for design development are becoming more aggressive. For example, a typical microprocessor life cycle ranges between three to four years (from design exploration to production); this has shrunk to less than a year for many IoT devices. Such aggressive contraction implies insufficient time for meticulous design review, potential misinterpretation of specifications and requirements, and consequent growth in errors. Furthermore, the reduced life cycle leads to less time for verification.

Consequently, verification engineers have to handle potentially more error-prone designs with limited time and resources. One consequence of this aggressive scheduling guided the emergence of in-field patching of devices and systems, potentially through software and firmware updates.

Today, many electronic devices exploit the System on Chip (SoC) design paradigm: the system is developed by integrating hardware elements referred to as Intellectual Property (IP). IPs are commonly developed and verified independently, either in-house or by a third party. Once the integration team receives the system specification for the final device, it collects all the necessary IPs. Typically each IP is designed to communicate through a standard interface such as an AMBA bus interface, SPI, or similar; this grants a smooth integration. From the verification point of view, there are two flows in the context of SoC design. The first involves the verification of each IP, while the second is intended to verify the fully assembled system. Considering modern devices' complexity, IP and SoC verification flows today consist of elaborate procedures requiring detailed planning covering the entire design life cycle. Pre-silicon verification is the significant resource-intense verification action during (and after) hardware development. It is a continuous process, with increasing maturity and complexity as the design evolves.

5.2 Challenges

Despite maturity, verification tools are not able to scale up to fulfill the needs of current SoC verification issues [133]. This section discusses some critical challenges due to the complexity (e.g., tool scalability) and some technology trends.

5.2.1 Reducing verification time

The exponential growth of the number of devices in the IoT paradigm has drastically shrunk the system development life cycle, leaving a short time for customized verification steps. However, each device features different functionality, performance, power, and security constraints. Consequently, the need for standardized, reusable verification flows that can be easily adapted to a diversity of devices arises.

Two possible approaches exist. The first consists of improving tool scalability to ease the verification phase, however, this objective remains elusive. The second approach aims at making the verification systems highly configurable. The same design can be configured (both from the hardware and software point of view) to work under several conditions.

However, developing a system with such high configurability is impractical; it is impossible to determine all the different use cases in advance. It would also mean monstrously increasing the system's complexity, making verification more complicated.

5.2.2 Limited tool scalability

As already introduced in the previous section, scalability remains a critical problem for verification technology. The increasing configurability and design complexity strongly limit the tool's scalability.

A growing trend, especially in formal methods, targets specific applications (e.g., security, etc.) rather than entire functional correctness. Moreover, the cost of simulation-based verification is reaching prohibitive levels as the architecture size increases. In addition, specific coverage tests can be prohibitive concerning the time/plan effort required.

5.2.3 Specification drafting

A fundamental challenge in nowadays verification is the lack of specifications. Typically, specifications rely on requirements documents, which can be ambiguous or vague under some design behavior.

While sometimes such omissions can be intentional, often the ambiguity is due to natural languages. The problem becomes significantly more aggressive in the context of non-ISA-based SoC designs than for classic microprocessors.

Although the ISA semantics are complex, their standardization and durability across product generations represent a clear understanding of the intended behavior of microprocessors. For ISA-based architecture, for example, it is widespread to have a simulator that can serve as a golden model for comparison. On the

other hand, for beyond-ISA-architecture is not so straightforward to characterize the intended behavior. SoC design requirements are composed of system-level requirements documents, integration documents, high-level-architecture documents, and microarchitecture documents, which are very complicated to reconcile.

5.2.4 Use case identification

Due to the continuous scaling of time-to-market, verification now tends to focus on a small number of use cases instead of verifying all possible scenarios.

For example, the intended usage for a low-power and low-performance device includes scenarios where different elements repeatedly transition into various sleep modes but would not enclose sustained high clock speeds execution.

Unfortunately, while reducing verification effort, this approach might add considerable complexity to the process. In particular, the usage scenarios could involve a complex interaction of hardware, firmware, and software. Furthermore, the SoC design and IPs have orthogonal verification requirements, flows, and timelines.

5.2.5 Security and functional safety

Security and privacy are necessary requirements for electronic devices nowadays. The main reason is that in IoT, millions of devices are connected to each other, devices that were not originally designed to communicate.

Therefore, security threats remain unclear. In addition, functional safety, i.e., the warranty that the device does not affect the environment in case of failure, is a critical requirement for sensitive applications such as aerospace or automotive. This consideration leads to fault-tolerant devices that embed modular redundancy to be resilient to failures.

5.3 Verification trends

The most recent trends observed in recent years can be summarized as follows:

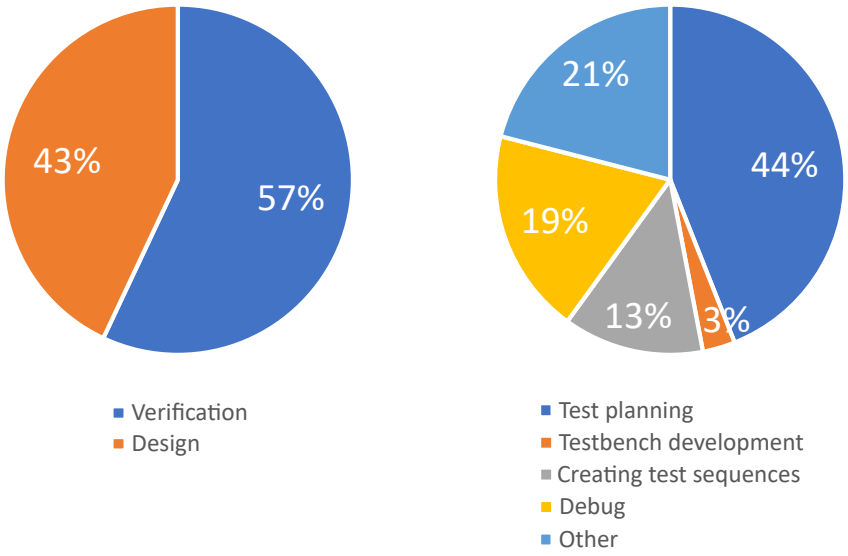


Fig. 5.1 Design vs. verification effort(left), verification breakdown (right).

- Verification now characterizes the bulk of the system design effort, representing about 57% [133] of the entire development time, as shown in Figure 5.1, left. Considering verification in detail, it can be seen that most of the time is spent on planning, crucial in fact to prepare an ad hoc strategy. Debugging the verification itself also represents a significant effort, Figure 5.1, right.
- we see an increase in the use of emulation models and FPGAs, especially for very complex designs. Complex SoCs, in fact, require a fast prototyping environment. Also, software or firmware plays a crucial role in modern systems, so emulation is critical.
- Most devices are produced after an average of two rounds of silicon. The first is usually used to detect all hardware problems, while the second is for interaction with the software. This again underscores the critical role of pre-silicon verification, especially simulation-based verification and targeted formal verification activities.

Current research trends focus verification on specific application domains, such as automotive or security, and then improve verification efficiency through data analytics. Rather than achieving complete functional coverage, they focus on critical design features.

5.4 Simulation UVM-based Testbench: basic structure

Simulation-based verification is performed through a test environment, an abstract model that drives stimulus over a Design Under Test (DUT). Functional verification consists of verifying the functionality of the IP, in other words, it has to demonstrate that the design implementation meets the specification.

In hardware verification, the test environment features a golden model (or reference model) of the specification that is compared against the DUT. As described before, the time required to develop a test environment is proportional to the design's complexity. Frequently, most of the verification time is consumed on assembling a testbench architecture. Also, it is essential to choose the proper verification strategy.

in observance of what we said above, a testbench must have the following characteristics:

- Modularity, the ability to decouple and reintegrate elements of the verification solution. This property reduces the system's complexity and allows the various units to be simplified.
- Scalability, scaled to the size of the project.
- Reusability, The possibility to re-use the environment components for other projects. This property drastically downsizes verification time and effort.
- Configurability, The ability to adapt the behavior of the environment with minimal input.

The Universal Verification Methodology (UVM) is the most popular in SoC functional verification. The UVM package includes SystemVerilog libraries used for the industry-grade verification environments. The UVM exploits features and design practices from Object-oriented programming, such as:

- Encapsulation: for modularity;
- Data Hiding: for abstraction;
- Generic Programming: for reusability;

- Inheritance: for scalability.

In the following, we will describe the basic UVM testbench architecture used to verify the design presented in the previous chapter.

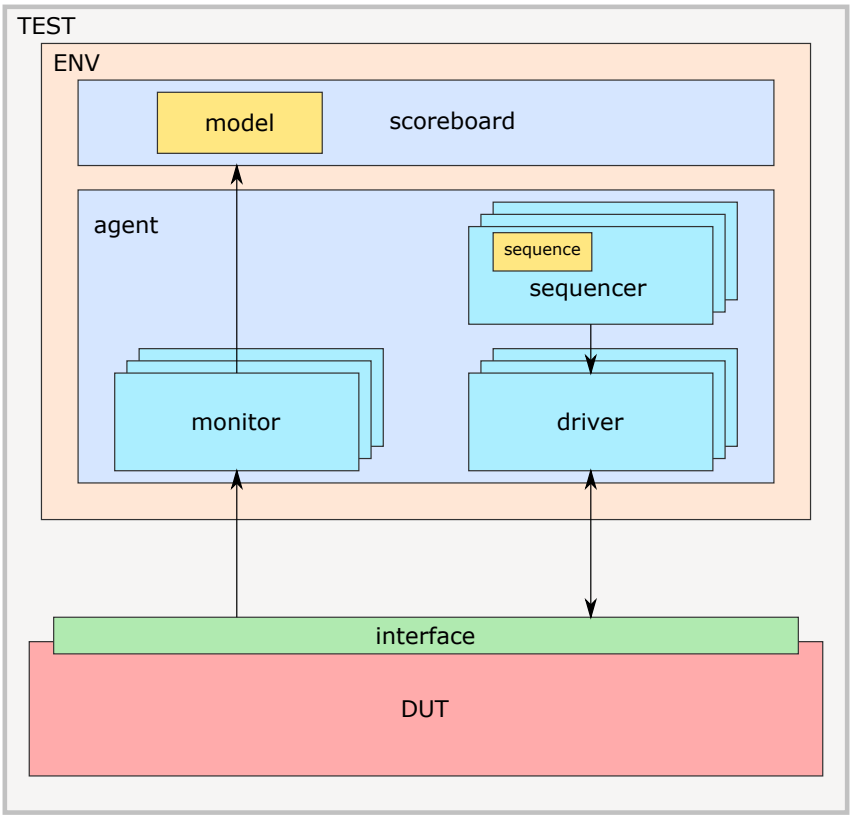


Fig. 5.2 UVM-based testbench architecture.

In Hardware Description Language (HDL), a traditional testbench is a wrapper code of the DUT. The Testbench top is a module that instantiates all the verification sub-components such as test, interfaces, and the DUT. An overview of a classic TB architecture is illustrated in Figure 5.2.

All items included in the TEST (gray) box are part of the classic UVM-based testbench and are written in SystemVerilog. The test has to instantiate the IP environment and provide constraints for simulation. In the environment, we find the scoreboard and the agent. The latter contains the monitors to capture the transactions on the various Device Under Test (DUT) interfaces. The sequences that a sequencer will provide to the relative driver will adequately drive the reference RTL interface

with the stimulus. The scoreboard is responsible for comparing the outputs produced by the RTL and the model and checking all interfaces for unused data.

5.4.1 Test

The Test class instantiates the environment and the sequences which will apply the stimulus. During verification, many test cases verify various targets under different scenarios. Rather than duplicate the same code for these tests multiple times, multiple tests are created by extending from a base class. Then, all the test classes derived from this base Test class can alter configurations and produce different tests. This is a clear example of modularity, scalability, re-usability, and configurability by the UVM. In addition, considering complex designs, each submodule is tested separately with its environment. When the complete core is tested, the core environment and all submodule environments are instantiated in the core test class, so there is redundancy in verification. In fact, it will be able to test the behavior of the core and each submodule decoupled at the same time.

5.4.2 Environment

The Environment clusters together all the related verification elements. The components instantiated in the Environment are the scoreboard and the agent. The Environment class also sets the connections between the agent's monitors and the scoreboard model FIFOs, which keep track of the incoming transactions.

5.4.3 Scoreboard

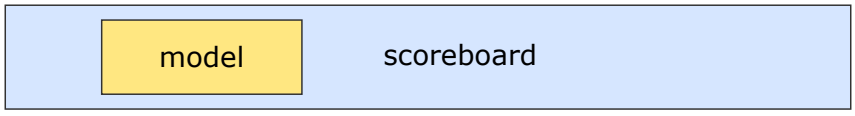


Fig. 5.3 Scoreboard and golden model.

The Scoreboard features queues (UVM FIFOs) that compare the expected (from the model) and actual (from the DUT) outcomes. This verifies the DUT's behavior. The Scoreboard acquires data in the queues through the respective monitor analysis

ports. The Scoreboard performs an element-wise comparison between the data contained in the FIFO queues and the ones produced by the model, present in Figure 5.3. An error is raised in case of any mismatch.

5.4.4 Agent

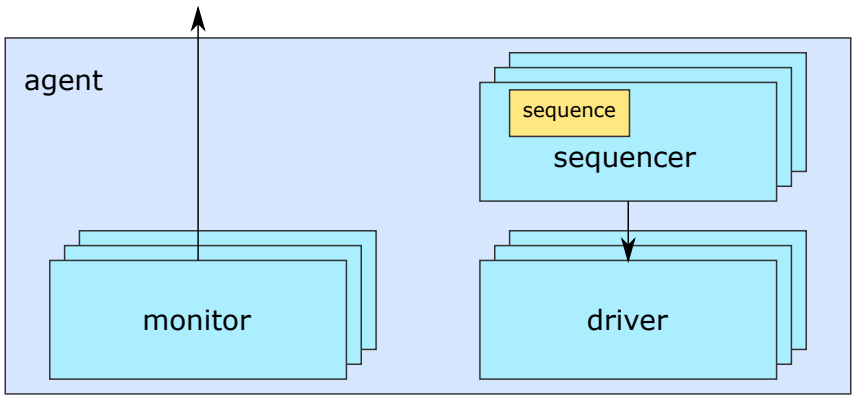


Fig. 5.4 Testbench agent, composed of monitors, drivers and sequencers.

The Agent encapsulates verification components that cope with the interface: the drivers, the monitors, and the sequencers, as shown in Figure 5.4. After the instantiations of these elements, it connects the sequencers with the relative drivers.

5.4.5 Sequencer and sequence

The sequencer manages the flow of sequences, a constrained random or directed data collection of sequence items. A sequence can be considered a data packet representing a transaction or protocol. Basically, a sequencer passes a transaction to the driver.

The Sequence is responsible for randomizing data in the transaction type format. It represents the core stimuli of a TB. A sequence item is the single element of a sequence, and it consists of information fields required to cause a stimulus.

5.4.6 Driver

The driver is generally responsible for driving the interfaces based on a specific protocol with a transaction (sequence). The driver can also transmit a reply to the sequencer if required.

5.4.7 Monitor

The monitor captures the transactions based on the protocol and sends these to the scoreboard via analysis ports. In addition, it can verify the protocol's correctness. All the above components, the driver, monitor, etc. are examples of UVM reusable components. They can be reused whenever the DUT exploits the same protocol interface.

5.4.8 DUT

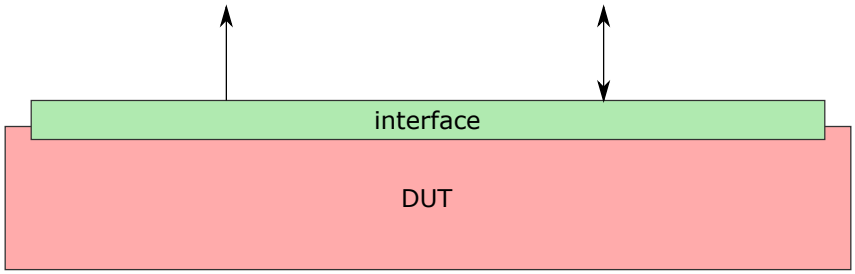


Fig. 5.5 Design Under Test.

The DUT represents the IP to be tested. At the verification level, all that is known is the specification of that element and its interfaces. Through the specifications, it is possible to replicate its behavior in the model contained in the scoreboard, while the interfaces are the access points for providing stimuli and obtaining responses, as shown in Figure 5.5. These outcomes will then be compared with those in the model.

5.5 Conclusion

Functional verification has become an essential aspect of the integrated circuit design process. As a result, significant resources are devoted to bridging design complexity

and verification efforts in industry and academia. As previously explained, many of the issues are related to lack of specification, tool scalability, time to market, and security. However, as the device ecosystem changes, moving more and more away from ISA-based devices toward devices beyond ISA, as a result of AI advancement that requires effective use of accelerators, verification becomes more and more of a bottleneck.

As studies demonstrate, verification is becoming an issue both in time and effort to the entire design development, without mentioning the cost. The tools used today do not scale perfectly with the complexity of the design. In addition, generating satisfactory tests that cover all cases of interest, including corner cases, is a process that requires planning and much effort on the part of engineers. Therefore, verification will become an increasingly integral part of the design process in the future, going so far as to condition the specifications themselves, forcing a design prone to verification.

The verification method by simulation built on the basis of a UVM testbench with an architecture similar to the one presented in this chapter was of fundamental use for the proper verification of the architectures presented in Chapter 4. While for the first configuration of the SMAC engine (without sparsity), the verification was pretty straightforward, i.e. the only need was to verify the correct execution of the convolution and communication protocols with the external TCDM memory, in the sparsity case, the verification got more complicated. Here, in fact, the verification was essential to debug the structure. As a starting point, the loading data process, which involves the local memory of weights and sparsity map and activations, is complex and must follow a specific sequence.

However, the most important critical issues to be verified were the correct alignment of the data, that is, the generation of the addresses of the weights with respect to non-zero activations and the accumulations between different SMACs. In fact, for the latter, it should be considered that the multiplication pattern was no longer deterministic as in the first case without sparsity, but each SMAC works as a separate unit with its own timings. This generated many bugs that were solved only by judicious verification work.

Chapter 6

Conclusions and Future Works

This thesis analyzed the challenges of accelerating the implementation of modern convolutional neural networks on embedded systems, both from a network model and hardware design perspective. We firmly believe that on-device inference and (partial) training of CNNs can pave the way for new applications characterized by low latency and high power efficiency standards. Therefore, the research challenges addressed in this thesis were motivated by the practical limitations of the embedded systems: the lack of computing capability, the minimal memory resources, and the demand for drastic energy efficiency. Overcoming these limitations relies on using quantized networks and a vertical optimization encompassing both the model and the hardware device, which, as demonstrated, can leverage different application domain-specific data features for more efficient acceleration.

Chapter 2 introduces a crucial background part related to architectural exploration for Deep Learning acceleration. In particular, it discusses the differences between temporal architectures (mainly CPU and GPU) and spatial architectures. In addition, it describes the use of quantized models and compression strategies to deal with sparse data representation. The importance of the architectural exploration described in this chapter lies in the ability to exploit a given feature (or features) of the network model or the data itself to derive an optimized strategy. For example, exploiting a given data characteristic such as sparsity allows for accelerated execution if all the operations that do not contribute to the final result are discarded. The critical message of this chapter is that starting from a domain-specific application, it is possible to derive several data flows and data properties considerations that,

through architectural exploration, can lead to highly optimized hardware solutions. These implementations are extremely valuable in embedded device scenarios where the computation resources are minimal. This chapter lays the foundation for the reasoning that will lead to the optimizations presented in subsequent chapters.

Chapter 3 combines the features of the Extreme Learning Machine with Continual Learning. In particular, after carefully analyzing ELM and its variants (OS-ELM and PE-ELM), we adopted the Online Sequential ELM to solve the challenges proposed by CL in the Single Incremental Task scenario. The SIT requires the neural network to be able to continuously learn as new batches of samples become available over time without forgetting the previously learned tasks (catastrophic forgetting). This also implies that the final layer, operating as a classifier, is expanded whenever a new class is encountered. In order to obtain fair comparisons with state-of-the-art, benchmarking was defined by considering two networks such as CaffeNet and GoogLeNet trained in SIT on CORe50. The two network models were suitably modified in the last layers to allow the inclusion of the OS-ELM. This approach shows encouraging results, for instance, in the case of CaffeNet our technique outperformed AR1, the current state-of-the-art in accuracy, reaching an accuracy of 47%. OS-ELM features two essential skills that should be considered in addition to the accuracy it can achieve. In fact, its flexibility allows it to learn from batches of any size, even with single-element batches, which is critical for a CL scenario. Moreover, regardless of the order of the batches encountered, the result in accuracy will always be the same.

Chapter 4 is divided into two main parts. In the first, a bit serial architecture is proposed for accelerating the inference of CNNs called the SMAC engine. This architecture is embedded in a heterogeneous platform for embedded devices. In the second part, the same architecture is revisited to accommodate the possibility of exploiting sparsity of activations to avoid operations between null elements. The Serial-MAC-engine (SMAC-engine) is a 65 nm convolutional hardware accelerator with variable parallelism suitable for quantized neural networks. The 14.28 GMAC/s at the cost of 0.58 pJ/MAC @ 1.0 V is achieved by exploiting a serial strategy where multiplication is performed bit by bit, immensely reducing the multipliers silicon area and consequently boosting the maximum accessible frequency. Furthermore, the vast selection in the data parallelism enables the architecture to run inferences on several applications and networks targeting the Internet-of-Things paradigm.

The architecture is then further developed to take advantage of the sparsity of activations to improve performance. However, the Sparse MAC engine requires a new data flow and a revised structure, introducing significant overhead in terms of area and energy. However, the throughput results improved almost by a factor of 2x over the previous implementation reaching peaks of 56.88 GMAC/s for Pa=8 and Pw=4 and sparsity 80%

The accelerator is fully embedded into a microcontroller-based platform, namely PULPissimo.

Chapter 5 introduces the verification method by simulation based on a UVM testbench. The testbench architecture presented in this chapter was of fundamental use for properly verifying the architectures presented in Chapter 4. For example, verifying the SMAC engine was fundamental to prove the correct execution of the convolution and communication protocols with the external TCDM memory. For the sparse SMAC engine, the verification got more complicated. Ensuring synchronization between SMACs working on different input rows requires a complex system of control units. This chapter aims to give an insight into how verification has been accomplished and discuss the challenges and future trends dictated by the verification of non-ISA-based architectures.

In summary, this thesis presents a detailed analysis of how to optimize and accelerate the execution of a CNN from both the model and implementation perspectives through the implementation of a digital architecture. The main contributions of this work are a new methodology that combines ELM and CL to achieve a generic methodology for continuous CNN learning, and an accelerator architecture that combines bit-serial and sparsity, the first of its kind.

6.1 Future Perspectives

Generally speaking, the complexity of DNNs is increasing over time by requiring more and more memory and computing power to run. This plays to the disadvantage of embedded applications. However, on the hardware side, devices continue to follow Moore's law, providing gradually more complete and higher-performing devices. This does not detract from the fact that optimization and design exploration techniques are and will be critical to the development of low-power architectures.

Chapter 3 future developments would be to combine OS-ELM and fine-tuning techniques to improve network accuracy. Since the OS-ELM trains only the last classification layer, using fine-tuning to redistribute learning over part of the layers of the network could improve the ability to extract meaningful features already in the first layers. In addition, extreme learning techniques have recently been developed for convolutional layers. Combining these techniques with continual learning could be further effective and ensure better distributed learning on all layers.

Regarding the SMAC engine, both the basic and the sparse versions, two main improvements are possible. One concerns the ability to support residual connections by giving the possibility to preload the input feature map in the AC3 accumulator registers. The second would be to enhance the power saving by clock-gating the internal accumulators of AC2 and AC3. Indeed, proper use of such a technique may reduce the dynamic power consumption by avoiding spurious commutations at the inputs of the unused registers. It is also possible to try new combinations of the data flow and expand the size of the matrix of SMACs. Finally, combine sparsity of weights and activations to completely eliminate unnecessary operations.

References

[1] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. IEEE Access, 8:225134–225180, 2020.

[2] Scott Freeman and Healy Hamilton. Biological Science, 2005.

[3] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 1–9, 2015.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 770–778, 2015.

[5] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 2261–2269, 2016.

[6] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 7132–7141, June 2018.

[7] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. In Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, page 3859–3869, Red Hook, NY, USA, 2017. Curran Associates Inc.

[8] 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 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), pages 92–104, June 2015.

[9] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In 2013 IEEE 31st International Conference on Computer Design (ICCD), pages 13–19, Oct 2013.

[10] L. Cavigelli and L. Benini. Origami: A 803-gop/s/w convolutional network accelerator. IEEE Transactions on Circuits and Systems for Video Technology, 27(11):2461–2475, Nov 2017.

[11] Vincenzo Lomonaco and Davide Maltoni. Core50: a new dataset and benchmark for continuous object recognition, 2017.

[12] Maurizio Capra, Francesco Conti, and Maurizio Martina. A multi-precision bit-serial hardware accelerator ip for deep learning enabled internet-of-things. In 2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), pages 192–197, 2021.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society.

[14] D. Zhang and S. Liu. Top-down saliency object localization based on deep-learned features. In 2018 11th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), pages 1–9, 2018.

[15] T. Treebupachatsakul and S. Poomrittigul. Bacteria classification using image processing and deep learning. In 2019 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), pages 1–3, 2019.

[16] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. CoRR, abs/2001.05566, 2020.

[17] M. Aibin. Deep learning for cloud resources allocation: Long-short term memory in eons. In 2019 21st International Conference on Transparent Optical Networks (ICTON), pages 1–4, 2019.

[18] H. C. Kaskavalci and S. Gören. A deep learning based distributed smart surveillance architecture using edge and cloud computing. In 2019 International Conference on Deep Learning and Machine Learning in Emerging Applications (Deep-ML), pages 1–6, 2019.

[19] R. Zanc, T. Cioara, and I. Anghel. Forecasting financial markets using deep learning. In 2019 IEEE 15th International Conference on Intelligent Computer Communication and Processing (ICCP), pages 459–466, 2019.

[20] J. J. Ying, P. Huang, C. Chang, and D. Yang. A preliminary study on deep learning for predicting social insurance payment behavior. In 2017 IEEE International Conference on Big Data (Big Data), pages 1866–1875, 2017.

[21] V. Ha, D. Lu, G. S. Choi, H. Nguyen, and B. Yoon. Improving credit risk prediction in online peer-to-peer (p2p) lending using feature selection with deep learning. In 2019 21st International Conference on Advanced Communication Technology (ICACT), pages 511–515, 2019.

[22] A. K. Arslan, Ş. Yaşar, and C. Çolak. An intelligent system for the classification of lung cancer based on deep learning strategy. In 2019 International Artificial Intelligence and Data Processing Symposium (IDAP), pages 1–4, 2019.

[23] Heba Mohsen, El-Sayed El-Dahshan, El-Sayed El-Horbarty, and Abdel-Badeeh M.Salem. Classification using deep learning neural networks for brain tumors. Future Computing and Informatics Journal, 3, 12 2017.

[24] C. Barata and J. S. Marques. Deep learning for skin cancer diagnosis with hierarchical architectures. In 2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019), pages 841–845, 2019.

[25] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. Journal of Field Robotics, Nov 2019.

[26] T. S. Li, P. Kuo, C. Chang, H. Hsu, Y. Chen, and C. Chang. Deep belief network–based learning algorithm for humanoid robot in a pitching game. IEEE Access, 7:165659–165670, 2019.

[27] C. Wang, D. Freer, J. Liu, and G. Yang. Vision-based automatic control of a 5-fingered assistive robotic manipulator for activities of daily living. In 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 627–633, 2019.

[28] J. Guan, W. Zhou, S. Kang, Y. Sun, and Z. Liu. Robot formation control based on internet of things technology platform. IEEE Access, 8:96767–96776, 2020.

[29] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. Ultra low power deep-learning-powered autonomous nano drones. CoRR, abs/1805.01831, 2018.

[30] 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 2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC), pages 1–7, 2019.

[31] C. Heghedus, A. Chakravorty, and C. Rong. Energy load forecasting using deep learning. In 2018 IEEE International Conference on Energy Internet (ICEI), pages 146–151, 2018.

[32] Maurizio Capra, Riccardo Peloso, Guido Masera, Massimo Ruo Roch, and Maurizio Martina. Edge computing: A survey on the hardware requirements in the internet of things world. Future Internet, 11(4), 2019.

[33] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, 49:269–284, 02 2014.

[34] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5:127–147, 1943.

[35] Frank Rosenblatt. The Perceptron: A Perceiving and Recognizing Automaton (Project PARA). Report No. 85-460-1. Cornell Aeronautical Laboratory, 1957.

[36] Yoshua Bengio. Learning deep architectures for AI. Foundations and Trends in Machine Learning, 2(1):1–127, 2009.

[37] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86:2278 – 2324, 12 1998.

[38] David H. Hubel and Torsten N. Wiesel. Receptive fields of single neurons in the cat's striate cortex. Journal of Physiology, 148:574–591, 1959.

[39] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, Proceedings of the 32nd International Conference on Machine Learning, volume 37 of Proceedings of Machine Learning Research, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[40] Ning Qian. On the momentum term in gradient descent learning algorithms. Neural Networks, 12(1):145–151, 1999.

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

[42] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res., 12(null):2121–2159, july 2011.

[43] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. CoRR, abs/1212.5701, 2012.

[44] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. International Conference on Learning Representations, 12 2014.

[45] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In Proceedings of the Twenty-First International Conference on Machine Learning, ICML '04, page 78, New York, NY, USA, 2004. Association for Computing Machinery.

[46] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(56):1929–1958, 2014.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115(3):211–252, 2015.

[49] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[50] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 5987–5995, July 2017.

[51] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018, pages 8697–8710. IEEE Computer Society, 2018.

[52] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016, pages 2818–2826. IEEE Computer Society, 2016.

[53] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In Satinder P. Singh and Shaul Markovitch, editors, Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA, pages 4278–4284. AAAI Press, 2017.

[54] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R. Manmatha, Mu Li, and Alexander J. Smola. Resnest: Split-attention networks. CoRR, abs/2004.08955, 2020.

[55] Tal Ridnik, Hussam Lawen, Asaf Noy, and Itamar Friedman. Tresnet: High performance gpu-dedicated architecture. CoRR, abs/2003.13630, 2020.

[56] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. Transforming auto-encoders. In Proceedings of the 21th International Conference on Artificial Neural Networks - Volume Part I, ICANN'11, page 44–51, Berlin, Heidelberg, 2011. Springer-Verlag.

[57] Geoffrey E. Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018.

[58] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part I, volume 11205 of Lecture Notes in Computer Science, pages 19–35. Springer, 2018.

[59] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, volume 97 of Proceedings of Machine Learning Research, pages 6105–6114. PMLR, 2019.

[60] D. Al-Dabass, P. Vindlacheruvu, and D. J. Evans. Parallelism in neural nets. Parallel Algorithms and Applications, 11(3-4):169–185, 1997.

[61] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12):2295–2329, Dec 2017.

[62] R James. Intel avx-512 instructions. Intel, 06 2017.

[63] BFLOAT16 – Hardware Numerics Definition, 2018.

[64] NVIDIA TESLA V100 GPU ARCHITECTURE, 2017.

[65] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. CoRR, abs/1410.0759, 2014.

[66] https://developer.nvidia.com/gpu-accelerated-libraries.

[67] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule (France), Oct 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[68] A. Vasudevan, A. Anderson, and D. Gregg. Parallel multi channel convolution using general matrix multiplication. In 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 19–24, 2017.

[69] Volker Strassen. Gaussian elimination is not optimal. Numer. Math., 13(4):354–356, aug 1969.

[70] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In Stefan Wermter, Cornelius Weber, Wlodzislaw Duch, Timo Honkela, Petia D. Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa, editors, Artificial Neural Networks and Machine Learning - ICANN 2014 - 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings, volume 8681 of Lecture Notes in Computer Science, pages 281–290. Springer, 2014.

[71] Michaël Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. In Yoshua Bengio and Yann LeCun, editors, 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings, 2014.

[72] Shmuel Winograd. Arithmetic complexity of computations, volume 33 of CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, 1980.

[73] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 4013–4021, June 2016.

[74] Mark Horowitz. Energy table for 45nm process.

[75] 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 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–12, Oct 2016.

[76] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. IEEE Micro, 37:12–21, 2017.

[77] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops, pages 696–701, June 2014.

[78] V. Sriram, D. Cox, K. H. Tsoi, and W. Luk. Towards an embedded biologically-inspired machine vision processor. In 2010 International Conference on Field-Programmable Technology, pages 273–278, Dec 2010.

[79] 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 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, pages 53–60, July 2009.

[80] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Bell, Jeff Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. DNN dataflow choice is overrated. CoRR, abs/1809.04070, 2018.

[81] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, page 247–257, New York, NY, USA, 2010. Association for Computing Machinery.

[82] S. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H. Yoo. 4.6 a1.93tops/w scalable deep learning/inference processor with tetra-parallel mimd architecture for big-data applications. In 2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers, pages 1–3, Feb 2015.

[83] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, and et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16, page 26–35, New York, NY, USA, 2016. Association for Computing Machinery.

[84] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and et al. In-datacenter performance analysis of a tensor processing unit. SIGARCH Comput. Archit. News, 45(2):1–12, jun 2017.

[85] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits, 52(1):127–138, Jan 2017.

[86] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[87] 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 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pages 609–622, Dec 2014.

[88] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pages 10–14, Feb 2014.

[89] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In NIPS, 2015.

[90] Fengfu Li and Bin Liu. Ternary weight networks. CoRR, abs/1605.04711, 2016.

[91] 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 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 2704–2713, June 2018.

[92] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011, 2011.

[93] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. Journal of Machine Learning Research, 18(187):1–30, 2018.

[94] Yunhui Guo. A survey on methods and theories of quantized neural networks. CoRR, abs/1808.04752, 2018.

[95] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. IEEE Transactions on Neural Networks and Learning Systems, 29(11):5784–5789, 2018.

[96] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. CoRR, abs/1602.07360, 2016.

[97] Charbel Sakr and Naresh Shanbhag. Per-tensor fixed-point quantization of the back-propagation algorithm. In ICLR, 2019.

[98] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. CoRR, abs/1603.01025, 2016.

[99] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. Compressing deep convolutional networks using vector quantization. CoRR, abs/1412.6115, 2014.

[100] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, NIPS'15, page 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[101] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In Xianghua Xie, Mark W. Jones, and Gary K. L. Tam, editors, Proceedings of the British Machine Vision Conference 2015, BMVC 2015, Swansea, UK, September 7-10, 2015, pages 31.1–31.12. BMVA Press, 2015.

[102] Richard Wilson Vuduc and James W. Demmel. Automatic Performance Tuning of Sparse Matrix Kernels. PhD thesis, University of California, Berkeley, 2003. AAI3121741.

[103] Google. https://cloud.google.com/solutions/migration-to-google-cloud-transferring-your-large-da (accessed on 15 Otober 2020).

[104] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen. Convergence of edge computing and deep learning: A comprehensive survey. IEEE Communications Surveys Tutorials, 22(2):869–904, 2020.

[105] Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks, 2013.

[106] Stephen Grossberg. Adaptive resonance theory: How a brain learns to consciously attend, learn, and recognize a changing world. neural. Neural networks : the official journal of the International Neural Network Society, 37, 10 2012.

[107] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. Neural Networks, 113:54–71, 2019.

[108] Robert M. French. Catastrophic forgetting in connectionist networks. Trends in Cognitive Sciences, 3(4):128–135, 1999.

[109] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of Psychology of Learning and Motivation, pages 109–165. Academic Press, 1989.

[110] Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. Psychological review, 97 2:285–308, 1990.

[111] Davide Maltoni and Vincenzo Lomonaco. Continuous learning in single-incremental-task scenarios, 2019.

[112] Zhizhong Li and Derek Hoiem. Learning without forgetting, 2016.

[113] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence, 2017.

[114] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. icarl: Incremental classifier and representation learning, 2016.

[115] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning, 2017.

[116] Davide Maltoni and Vincenzo Lomonaco. Semi-supervised tuning from temporal coherence, 2015.

[117] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: Theory and applications. Neurocomputing, 70(1):489 – 501, 2006. Neural Networks.

[118] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

[119] N. Liang, G. Huang, P. Saratchandran, and N. Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Transactions on Neural Networks, 17(6):1411–1423, 2006.

[120] Yueqing Wang, Yong Dou, Xinwang Liu, and Yuanwu Lei. Pr-elm: Parallel regularized extreme learning machine based on cluster. Neurocomputing, 173, 09 2015.

[121] Yuxin Wu and Kaiming He. Group normalization, 2018.

[122] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018.

[123] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. Pulp: A parallel ultra low power platform for next generation iot applications. In 2015 IEEE Hot Chips 27 Symposium (HCS), pages 1–39, Aug 2015.

[124] Sayeh Sharify, Alberto Delmas Lascorz, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. CoRR, abs/1706.07853, 2017.

[125] Francesco Conti, Robert Schilling, Pasquale Davide Schiavone, Antonio Pullini, Davide Rossi, Frank Kagan Gürkaynak, Michael Muehlberghuber, Michael Gautschi, Igor Loi, Germain Haugou, Stefan Mangard, and Luca Benini. An iot endpoint system-on-chip for secure and energy-efficient near-sensor analytics. CoRR, abs/1612.05974, 2016.

[126] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. arXiv e-prints, page arXiv:1801.06601, Jan 2018.

[127] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors. arXiv e-prints, page arXiv:1908.11263, Aug 2019.

[128] 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 Journal of Solid-State Circuits, 53(4):983–994, April 2018.

[129] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. IEEE Journal of Solid-State Circuits, 54(1):173–185, Jan 2019.

[130] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1–12, 2016.

[131] Ya Le and Xuan S. Yang. Tiny imagenet visual recognition challenge. 2015.

[132] Malay Ganai and Aarti Gupta. SAT-Based Scalable Formal Verification Solutions (Series on Integrated Circuits and Systems). Springer-Verlag, Berlin, Heidelberg, 2007.

[133] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. Challenges and trends in modern soc design verification. IEEE Design Test, 34(5):7–22, 2017.

Appendix A

List of Publications

This appendix contains a complete list of the publications co-authored by Maurizio Capra during the years of his PhD. This list also includes publications not presented in this dissertation.

- **Capra, M**.; Peloso, R.; Masera, G.; Ruo Roch, M.; Martina, M. Edge Computing: A Survey On the Hardware Requirements in the Internet of Things World. Future Internet 2019, 11, 100. https://doi.org/10.3390/fi11040100
- **M. Capra**, B. Bussolino, A. Marchisio, G. Masera, M. Martina and M. Shafique, "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead," in IEEE Access, vol. 8, pp. 225134-225180, 2020, doi: 10.1109/ACCESS.2020.3039858.
- **M. Capra**, F. Conti and M. Martina, "A Multi-Precision Bit-Serial Hardware Accelerator IP for Deep Learning Enabled Internet-of-Things," 2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2021, pp. 192-197, doi: 10.1109/MWSCAS47672.2021.9531722.
- **Capra, M**.; Bussolino, B.; Marchisio, A.; Shafique, M.; Masera, G.; Martina, M. An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks. Future Internet 2020, 12, 113. https://doi.org/10.3390/fi12070113

- Peloso, R.; **Capra, M.**; Sole, L.; Ruo Roch, M.; Masera, G.; Martina, M. Steerable-Discrete-Cosine-Transform (SDCT): Hardware Implementation and Performance Analysis. Sensors 2020, 20, 1405. https://doi.org/10.3390/s20051405

- Sole, L., Peloso, R., **Capra, M.**, Ruo Roch, M., Masera, G., Martina, M. (2020). VLSI Architectures for the Steerable-Discrete-Cosine-Transform (SDCT). In: Saponara, S., De Gloria, A. (eds) Applications in Electronics Pervading Industry, Environment and Society. ApplePies 2019. Lecture Notes in Electrical Engineering, vol 627. Springer, Cham. https://doi.org/10.1007/978-3-030-37277-4_16

- **Capra, M.**; Sapienza, S.; Motto Ros, P.; Serrani, A.; Martina, M.; Puiatti, A.; Bonato, P.; Demarchi, D. Assessing the Feasibility of Augmenting Fall Detection Systems by Relying on UWB-Based Position Tracking and a Home Robot. Sensors 2020, 20, 5361. https://doi.org/10.3390/s20185361

- Anzalone, E., **Capra, M.**, Peloso, R., Martina, M., Masera, G. (2021). Low-Power Hardware Accelerator for Sparse Matrix Convolution in Deep Neural Network. In: Esposito, A., Faundez-Zanuy, M., Morabito, F., Pasero, E. (eds) Progresses in Artificial Intelligence and Neural Systems. Smart Innovation, Systems and Technologies, vol 184. Springer, Singapore. https://doi.org/10.1007/978-981-15-5093-5_8