POLITECNICO DI TORINO

Master's Degree in Electronic Systems

Master's Degree Thesis

Approximating Deep Convolutional Neural Networks through Bit-level Masking of Network Parameters

Supervisors

Prof. Guido MASERA

Prof. Maurizio MARTINA

Prof. Muhammad SHAFIQUE

Project Ass. M. Abdullah HANIF

Candidate

Giuseppe Maria SARDA

IV 2020

Summary

In the past years, a wide field of research was focused on the possibility of exploring new solutions for pushing artificial intelligence from the edge of technology to cheap and handy platforms. Remarkable signs of progress have been reached, and machine learning appeared in people's everyday life.
This work aims to propose a new approach that allows deep convolutional neural networks to be helpful in more IoT contexts, where the computational capability is a limit, or even to alleviate constraints in data centers, where computers process a massive stream of information continuously. In detail, this thesis provides a way to speed up hardware design solutions. Timing surplus can be exploited to design architectures that require less power while providing very high precision in image classification and in many other problems.
Several techniques are available to improve such implementations. The cost is often either the drop of output accuracy or a time-exhaustive computation for tuning and re-training networks. In the first case, solutions can not be exploited for safety-critical applications where even a slight inaccuracy in output can result in unwanted effects; in the second case, the computational limit of many error-resilient applications, which cannot perform in-place re-training, represents a critical restriction.
This work collects positive contribution provided by all these solutions and proposes a new versatile approach that combines hardware-aware software tuning and approximate arithmetic, without performing any time-expansive computation, after a specific configuration is chosen.
The idea consists of manually forcing to zero, or *masking*, specific bits of the kernels inside networks' layers. Every layer will have sets of weights that are forbidden depending on the underlying hardware they will be mapped to. In this way, it is possible to commit an architecture where time constraints are more relaxed then reference models, and remaining slack time can be exploited for low energy/power designs.
For balancing masking operation and preventing any degradation on accuracy caused by parameters alteration, three low computational cost techniques are proposed: the *optimal fast substitution of weights* for minimizing the impact of

masking process, a *quantization-range correction* for excluding the last forbidden set of values and improving parameters' distribution, and the *neurons bias compensation* for masks' effect mitigation.
Choosing the optimal mask to apply layer-wise can improve the distributions of feature maps across the network significantly. However, since the number of possible configurations scales exponentially with the complexity of the model, it is unfeasible to explore the solution space thoroughly. A heuristic approach was preferred to minimize in parallel the accuracy drop caused by masks and the delays of dataflows. The NSGA-II genetic algorithm was exploited for this purpose. Masks fit as the genes and the mutations applied to neural network structures.
Thanks to the optimization, a close-to-true Pareto front of solutions can be obtained: which configuration should be preferred from this set, it is up to the designer's choice depending on the target application.
Both safety-critical and error-resilient scenarios can largely benefit from this simple flexible approach.

A Mattia e Chiara,
i compagni migliori che potessi
desiderare lungo il mio cammino.

Acknowledgements

I would like to thank first of all professor Guido Masera for directing me to this project at the technical university of Vienna and for encouraging my passion for hardware design during labs.
I want to express also my gratitude to Abdullah and professor Shafique for their professional support, that made this thesis work possible, helped me developing my skills and create awareness of what I want to achieve in my future.
Thank you Alessio, for being a colleague and a friend in Vienna. Thank you Antonio, Riccardo and Valerio for the awesome working time in low-power lab.
My family and all my friends assisted me since the first year of my academic record, your support has been priceless and fundamental. Thanks to all of you from deep within my heart and my soul.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

ANN
Artificial Neural Network

ASIC
Application Specific Integrated Circuit

CPU
Central Processing Unit

DL
Deep Learning

CNN
Convolutional Neural Network

DNN
Deep Neural Network

FP
Floating Point

IC
Integrated Circuit

GPU
Graphics Processing Unit

MAC

Multiply-Accumulate Operation

ML

Machine Learning

NSGA

Non-domnated Sorting Genetic Algorithm

PE

Processing Engine

RF

Register File

STE

Straight-Through Estimator

WAD

Weighted Average Delay

Introduction

> *Talos, the bronze man, ... was given by Zeus to Europe as guardian of the Island (Crete), which he walked down three times a day with his bronze feet. His body and limb were made of unbreakable bronze, but on his ankle, under the tendon, he had a blood vein, covered by a thin membrane, which was for him both life and death.*
>
> ---
>
> *Argonautica*
> *Apollonius of Rhodes*

Talos was a mythological giant automaton whose task was to protect the Crete island from pirates and invaders. We have traces of his myth since 400 B.C.
Even in ancient Greece, inventors have dreamed of creating the machine capable of thinking, of solving a task on his own with human-like intelligence. Today the will of having an artificial brain for everyday life support hasn't faded: Machine Learning is a very broad field that is gaining strength in many applications and it interests a wide number of researchers all over the world; artificial intelligence can safely recommend cesarean delivery, play difficult Atari games, and recognize most of the spoken languages on Earth. One of the most powerful branches of ML is *Deep Learning*.

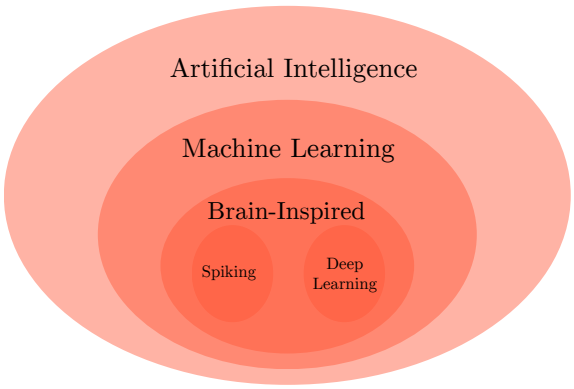

Figure 1: Deep learning in the wider context of artificial intelligence [1].

Deep convolutional neural networks are one of the most important tools created so far in the field of computer science. A vast spectrum of applications benefits from deep learning employment, which was amplified thanks to nowadays massive stream of data provided by a society always connected to the internet. In many contexts, computers overcame human precision and several modern studies aim to expand known approaches to medicine, economics, and environment areas.
The main advantage brought by this branch of artificial intelligence is the capability of networks to learn autonomously the fundamental features needed to complete a particular task. The programmer doesn't have to program the algorithm explicitly, but he has to designing a structure that adapts the most to the target application. The most critical aspect of DNNs is the computational load that has to be satisfied to achieve high levels of accuracy; deeper networks can map many more details, hence they perform remarkably better at the cost of powerful and energy-consuming platforms.
This work aims to propose a new approach that allows neural networks to be helpful in more IoT contexts, where the computational capability is a limit, or even to alleviate constraints in data centers, where computers process a massive stream of information continuously. More specifically, this thesis provides a way to speed up application specific hardware designs. Timing surplus can be exploited to commit architectures that require less power while providing very high precision in classification problems.
Some state-of-the-art works [2][3][4][5] show how DNNs' resilience to processing errors allows approximate computing solutions. Other researches have also exploited software tuning in light of underlying hardware [6]. These papers, even if they provide significant improvement in performance, they accept some output quality loss, which can't be tolerated in safety-critical applications where even a slight inaccuracy in output can result in unwanted effects.
Several techniques have been perfected to improve accuracy in such implementations. Hardware aware training and fine-tuning network's parameters are two examples that worked very well in many cases [7]. The cost is time-exhaustive computation: often approximate computing models require long simulation time, and, if combined with heuristic approaches for tuning configurations, they may lead to unfeasible waiting time. Moreover, in some contexts re-training is also not possible due to the lack of backward process implementation or because training set may not be available.
This work collects positive contribution provided by all these solutions and proposes a new versatile approach that combines hardware-aware software tuning and approximate arithmetic, without performing any time-expansive computation when a specific configuration is chosen.
The idea came from the possibility to analyze which set of weights, from each layer, triggers the slower paths during execution; if those values are then excluded from

the inference process, it's possible to design a contracted architecture that works at a faster frequency without incurring in any timing error. Systolic array of MAC units is the most common solution for deep learning acceleration and it will be the hardware at the base of this analysis.
Knowing which values are allowed, multiplier architecture can be redesigned with shorter critical paths. Accumulation is then performed with accurate arithmetic since addends can't be controlled a priori, and the literature demonstrated that approximating adders has a high impact on DNN accuracy[8]. For example, if one of the i-th bit of the control-operand for the multiplication is known at design time to be never 1, the circuitry required for performing the i-th partial product addiction can be dropped, allowing faster and equally accurate hardware. For pursuing this possibility, quantized network's layers are masked at evaluation time, forcing specific weight bits to zero.
For balancing masking operation and preventing any degradation on accuracy caused by parameters alteration, three low computational cost techniques are proposed:

- *optimal fast substitution* of weights for minimizing the impact of masking process
- *quantization-range correction* for excluding the last forbidden set of values and improving parameters' distribution
- neurons *bias compensation* for masks' effect mitigation

From the hardware side, the *carry-save tree multiplier* was chosen as the case study to analyze improvements on delays; all possible contracted configurations, allowed by the software approach, were automatically generated and synthesized for timing classification.
Choosing the optimal mask to apply layer-wise can improve the distributions of feature maps across the network significantly. However, since the number of possible configurations scales exponentially with the complexity of the model, it is unfeasible to explore the solution space thoroughly. A heuristic approach was preferred to minimize in parallel the accuracy drop caused by masks and the delays of dataflows. The NSGA-II genetic algorithm was exploited for this purpose. Masks fit as the genes and the mutations applied to neural network structures.
Thanks to the optimization, a close-to-true Pareto front of solutions can be obtained: which configuration should be preferred from this set, it is up to the designer's choice depending on the target application.
Both safety-critical and error-resilient scenarios can largely benefit from this simple flexible approach.
The drawback is that the size of the Pareto-front highly depends on how many models the genetic algorithm explores. An extensive heuristic analysis requires, of

course, some computational time.
The work is examined in this paper as follow: Chapter 1 quickly reviews deep learning and present the algorithm at the foundation of this work, Chapter 2 presents the most common hardware solutions and necessary procedures for increasing network sparsity, Chapter 3 presents related works that inspired the proposed approach, Chapter 4 describes masks from the software side and all the techniques that mitigate their effects on the classification accuracy, Chapter 5 shows results from hardware analysis, Chapter 6 pictures how the optimization problem was modeled and solved through the NSGA-II genetic algorithm; results are there also provided.

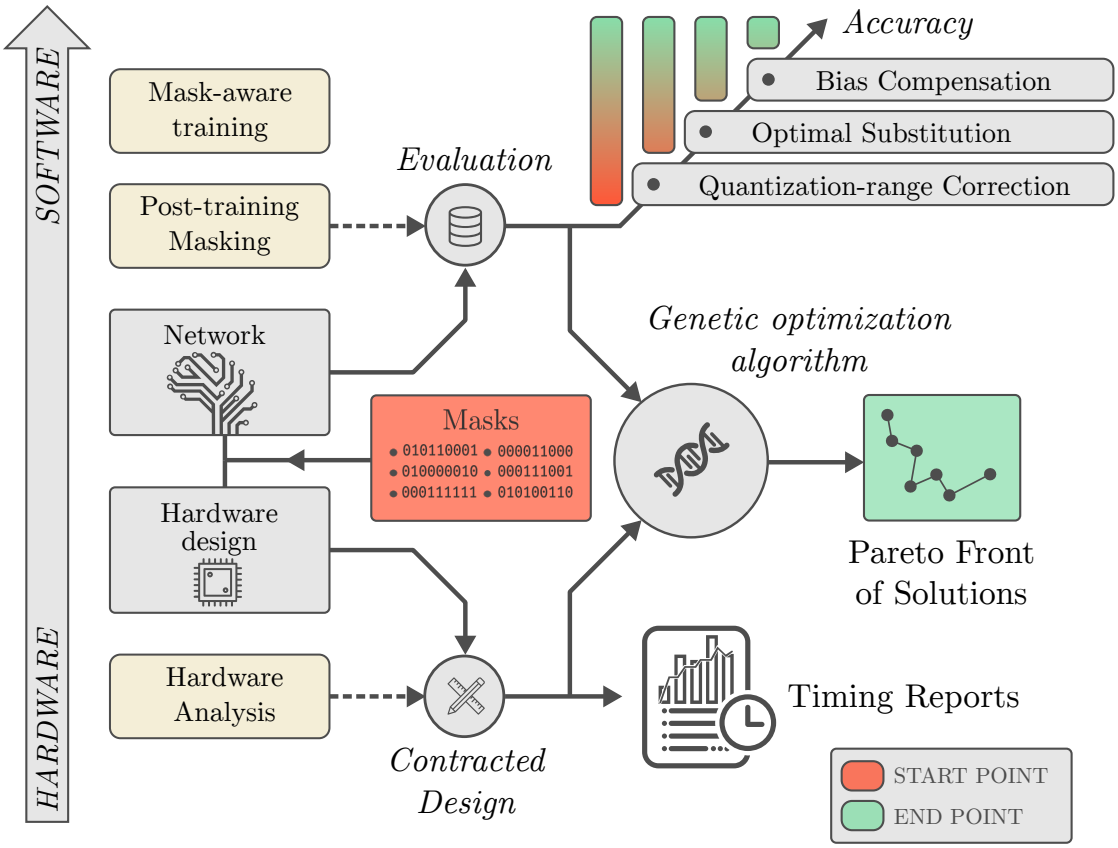

Figure 2: Overall scheme of the thesis work.

Chapter 1

A review on Deep Learning

According to *Goodfellow at al.* [9], machine learning is the only AI we have today to let computers act in complicated, real-world environments. Deep Leaning is a kind of learning that achieves excellent flexibility and robust deployment in many fields thanks to the capability to represent the world as a nested hierarchy of concepts autonomously.
DNNs are organized into sequential layers, every one different from the others; simple tasks are solved in the early stages, while complexity grows going more in-depth in the architecture. For understanding better this point, one can consider image recognition context: given a fixed number of objects, called classes, input images must be labeled according to what they represent. The neural network assigns a probability for each class that expresses how close is the processed data to that particular tag; this process may be done starting to detect edges in the image, then corners and so on with more complex shapes. After the first steps, object parts can be detected like ears, a nose, or eyes if "dog" is one of the possible labels. The complexity of feature analysis will continue to grow until the machine will be able to tell, with good accuracy, to which class the subject of the picture belongs. What makes this approach so powerful is that the programmer does not need to specify the features of each class, such as how many legs, arms, eyes each object has. But all these, and even more complex and abstract information are contained in the weights learned during training. Figure 1.1 shows the differences between programming approaches.
It is possible to think about this concept as data representation problem. Everybody is capable of solving faster arithmetic on Arabic numerals rather than on Roman numerals even if in the end the same operation has to be performed [9].
However, in many tasks, giving a correct representation or knowing which feature the machine should extract to make a correct prediction is not trivial. It is common to know how an eye looks like, but how is it possible to indicate it in pixel values? It is even harder if all possible variations in all possible pictures of a dataset must

be taken into account, since objects will appear cropped, rotated, flipped and so on.
In DNNs, this is not a problem, as it was already said: representation of classes is dissolved into the weights; a simple network example is provided by Figure 1.2.

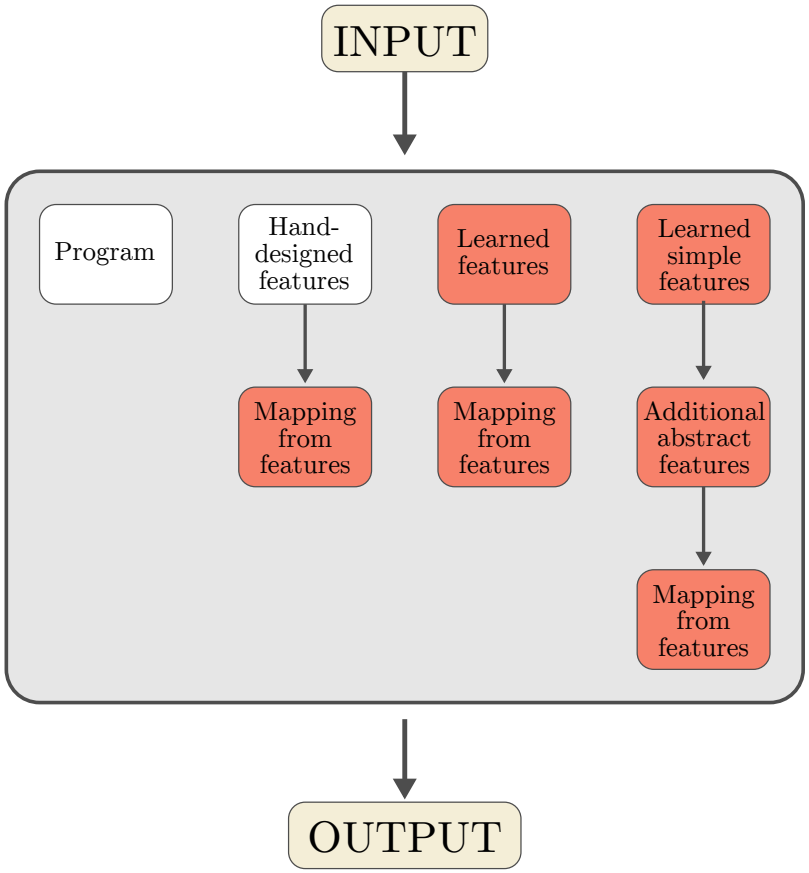

Figure 1.1: The graph shows the basic difference between approaches used for solving a task. The *leftmost* represent the classical programming paradigm. The *second* was firstly used for cesarean delivery recommendation, in which the programmer hard-coded features and users were giving several relevant information to the neural network, such as the presence of shocks or problems in the uterus [10]. The *third* refers to shallow networks in which only one hidden layer is used for retrieving object features. The *last one* represents DNN where deep architecture leads to a more abstract and complex representation of data [9]

.

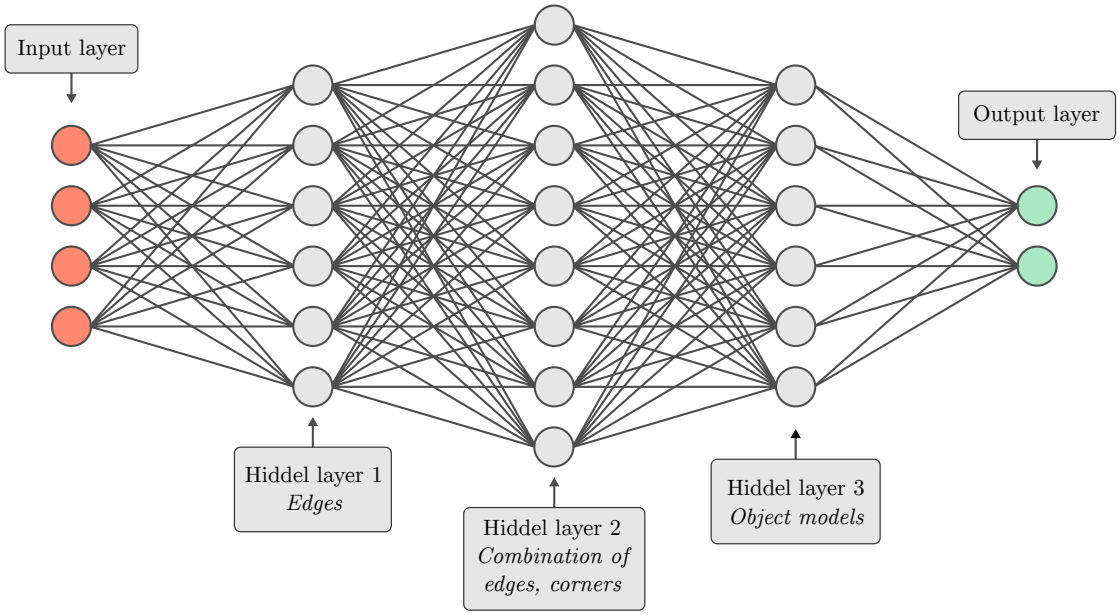

Figure 1.2: Typical architecture of a Deep Neural Network.

1.1 Artificial and human intelligence

Within machine learning, as shown in Figure 1, there is a trend indicated as brain-inspired. Since the best "tool" known so far for learning, analyzing, and solving problems is the human brain, these kinds of algorithms find inspiration on what we know about how human beings learn and execute tasks. There are many limits in this emulation since the technology used inside computers is fundamentally very different from the biological processes of living beings: models of artificial intelligence try to duplicate the functionality of the brain rather than providing a realistic shape of it. Greater neural realism has not led to improvements in machine learning performance.
Deep learning, as a subclass, finds inspiration from the essential computation element of the brain: *the neuron*.
Neurons are connected through two types of "channel":

- *dendrites* provide multiple input signals to the central elaboration system.
- *axon* carries a single result.

Input and output connections are usually referred to as *activations*. The link between one neuron axon and the next one's dendrite is called a *synapse*. Figure 1.3 pictures the artificial neuron. There are around 86 billion neurons and 10^{15} synapses on average in the human brain.

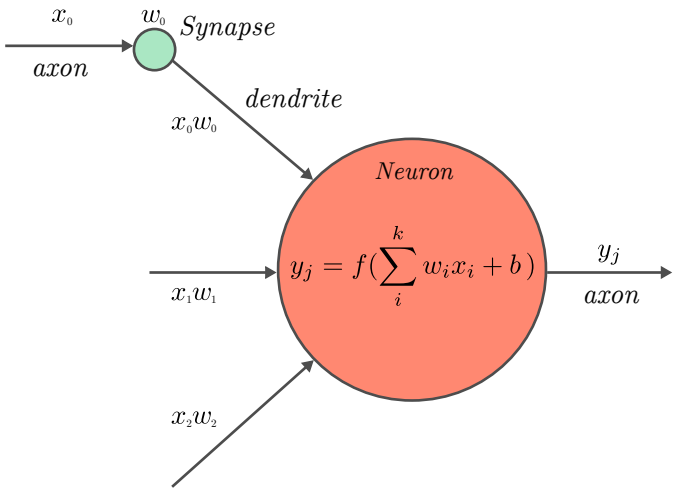

Figure 1.3: Scheme of the model of a neuron.

The main task of synapses is to scale the signal crossing itself, which can be seen from a mathematical point of view as a multiplication by a factor, called *weight*. Scientists believe that we learn by changing these weights associated with neurons' connection: different scale corresponds to a different response to inputs. All these signals, received by the central cell, are then summed and a non-linear transformation is applied to the result. Equation 1.1 resumes what has been just described.
This algorithm is then executed by every neuron or, for artificial intelligence, by every layer. As already said, a deeper structure corresponds to more complexity and abstraction. Today DNNs can have even up to one thousand layers.

$$y_j = f(\sum_i^k w_i x_i + b) \tag{1.1}$$

Scientists think that probably mammalians use a single procedure to complete everyday tasks; the brain is made of multiple units that, if analyzed alone, are just biological cells, but connected all together, they become intelligent. Similarly, some DNNs structure, made of primary PE, can be used in different applications.
For what concerns learning, there would always be a set of weight applied to the architecture that minimizes the distance between the expected behavior and the actual behavior. This problem can be solved with algorithms that find the minimum of a given n-variable cost-function. Most used algorithms today are modified versions of the stochastic gradient descent.
How the human brain applies his learning capability, it is today far to be known [11].

1.2 History

Despite what one may think, machine learning roots settle down in the middle of the last century. Three primary waves of interest characterized the topic: between the 1940s-1960s, the artificial intelligence research was known as *cybernetics*, then in the 1980s-1990s, it had the name of *connectionism*, while *deep learning* started in 2006. Early attempts on machine learning were simple linear models based on neuroscientific studies, in which weights of the network had to be set by hand by an operator [12].
At the end of the '60s, two important works set up the capability of learning weights, elaborating labeled data [13] [14]. Adaptive Linear Element (ADALINE) was the first one using stochastic gradient descent for finding the best network configuration; another breakthrough was reached with discovering that linear models did not match precisely the brain-behavior, non-linearity was added inside the artificial neuron for improving the overall outcome.
In the 90s, connectionism brought the idea of using a large number of simple computational units to achieve intelligent behavior when all connected. In this context, Geoffrey Hinton provided the concept of *distributed representations*: all input of a system should be represented by many features, where these features must not be valid only for those specific data used for training, but should be involved in the representation of all possible coherent inputs [15].
Other noteworthy accomplishments were the successfully use of the back propagation algorithm [16] for training, and the introduction of the long short-term memory for modeling long sequences of data [17], today used in many natural language processing tasks at Google.
Algorithms set up before 2000 were proved to work well, but all had a simple issue: they required computationally high cost, which was satisfied only with modern machines from 2006. "Deep learning" was used the first time for indicating the importance of depth in artificial intelligence, that could be achieved thanks to high-performance architectures.
Nowadays, the trend is to use supervised learning algorithms (e.g., providing labeled data) and to exploit the ability of DNN to leverage large datasets. The increasing amount of information was possible thanks to the digitization of society. Most of the activity in which people are involved is recorded. Facebook and Google lead in storing images, voice recordings and videos that can be labeled and used for machine learning research purposes. But deep learning extends its capabilities to medical, economic, weather forecasting and many more application fields, where today there is not a leader in the market.
Artificial network match or even exceed human performance when trained with datasets containing 10 million labeled examples, but there are important research

areas that focus on working with smaller datasets or unlabeled data through unsupervised or semi-supervised learning.

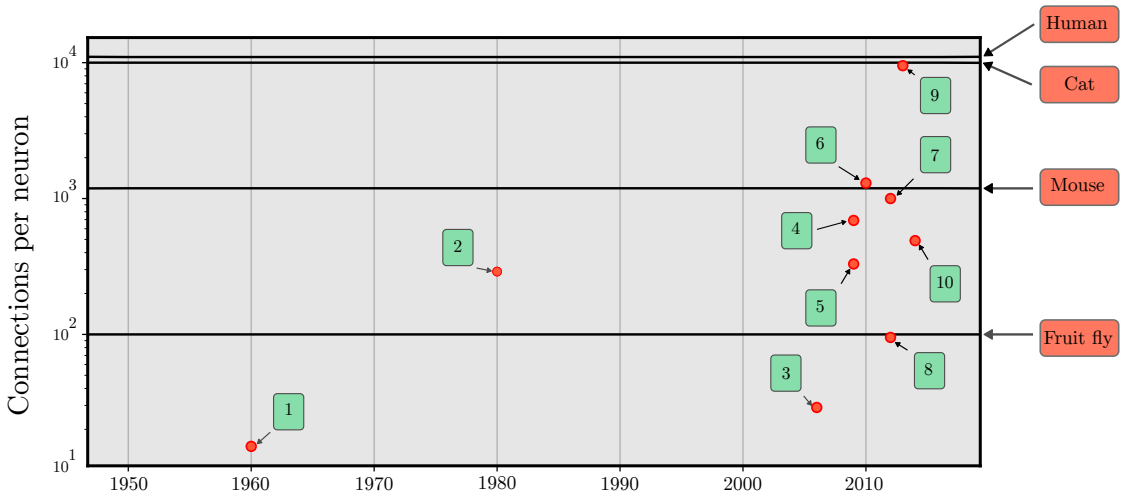

Figure 1.4: Number of connections between elementary units of neural network have been always limited by hardware capabilities. Here they are compared with the complexity of the brain of some living beings [18]

.

Here follow the references for data in the plot above:

1. Adaptive linear element [14]
2. Neocognitron [19]
3. GPU-accelerated convolutional network [20]
4. Deep Bolzmann Machine [21]
5. Unsupervised convolutional network [22]
6. GPU-accelerated multilayer perceptron [23]
7. Distributed autoencoder [24]
8. Multi-GPU convolutional network [25]
9. COTS HPC unsupervised convolutional network [26]
10. GoogLeNet [27]

From 2006, even if Moore's law slowed down because of technology limits, the performance of computational units are increasing in a close-to-exponential trend. This opens the possibility of deeper and broader neural networks that achieve higher accuracy on more complex tasks. The most complicated architecture today can be compared to the nervous system of frogs (Figure 1.4), which is already an important goal. By 2060, human brain complexity may be reached.
The impact of the progress of artificial intelligence is evident. ImageNet Large Scale Visual Recognition Challange is every year won with a lower top-5 error [28]. Alex-net scored 15.3% in 2012 [25] while today the record is 3.6%. In pedestrian detection and traffic sign classification, AI reached superhuman accuracy in the context of autonomous driving; sequence of characters from an image can be easily detected, rather than just identifying a single object. The most recent application includes even prediction on how molecules may interact in the pharmaceutical design of new drugs [29].
What can be said is that deep learning is gaining strength in many more applications and years ahead will be full of challenges to bring it to another level, delivering improvements in many aspects of everyone's everyday life such as security, health, and assistance.

1.3 The Deep Learning algorithm

Supervised deep learning algorithm, as already said, is one of the most important and well employed in the machine learning field. It opens up many deployment contexts, supporting hardware solutions for better performance. This is the one chosen as the case study for finding improvements in this thesis work.
For a better understanding of the matter, here follows a fast review of the basic recipe selected as the starting point for the analysis.
Only *training* (Algorithm 1), that is the procedure through which the network learns, will be examined. Processing test or real-world data is referred to as *inference*; this operation is also performed during the off-line training, a stand-alone explanation is hence omitted.
DNNs come in different shapes and sizes according to the application are designed for. Input will be a set of values representing the information to be elaborated. Image-classification networks are used as in the following chapters for studying any possible improvements.
Before beginning the elaboration, training data must be fetched. Networks are designed to work for specific inputs, even though they can keep almost the same inner structure with different datasets. For instance, if an AI is supposed to label 28x28 pixels black and white images, then it cannot be used for 32x32 ones.
The most famous dataset is the MNIST database [30] (Modified National Institute of

Algorithm 1 Deep Neural Network training algorithm

```
epoch=0
while epoch < EPOCH_NUMBER do
    batch_count=0
    while batch_count < TOTAL_MINIBATCHES do
        data, labels = fetchData(dataset, batch_count)
        predictions = forwardpass(batchData)
        loss = lossFunction(predictions, labels)
        gradiants = backwardpass(loss)
        network.weights = optimizer(network.weights,gradients,learning_rate)
        batch_count = batch_count + 1
    end while
    epoch = epoch + 1
end while
```

Standards and Technology, Figure 1.5). It is an extensive collection of handwritten digits, which has been used for training and benchmarking many neural networks in the past. Today it does not represent a challenge anymore since it is common to reach very high accuracy with simple architectures. An alternative was provided by Zalando with the Fashion MNIST database [31], a new collection in the exact frame of the first one but with pictures of clothes instead of numbers. It keeps the same amount of samples (50000 for training and 10000 for testing) of the same shape, divided into ten different classes.

Other well-known datasets are the CIFAR-10 and the CIFAR-100 that respectively are labeled into 10 and 100 classes[32]. While the one used today for the Large Scale Visual Recognition Challenge is Image-Net [28].

Once data is fetched, the forward-pass is performed. This step is identical to a normal inference process. A sequence of arithmetic operations is repeated between different layers, which do not keep any memory of previous evaluations.

Neurons between stages may be *fully connected* between each other, or in case some links are missing, the stage will be referred to as a *sparsely connected layer*. The operation performed by each unit is a convolution of the results obtained by the "neighbors" that are connected to it. Convolution consists of the weighted sum of each input data; weights remain always the same for processing a specific output feature map of a layer. Figure 1.6 and Figure 1.7 show an example of high dimentional convolution for the LeNet network, used for MNIST dataset classification.

After each section, other transformations can be performed on data for improving network performance or for reducing computational load, this depends mainly on the designer's choice.

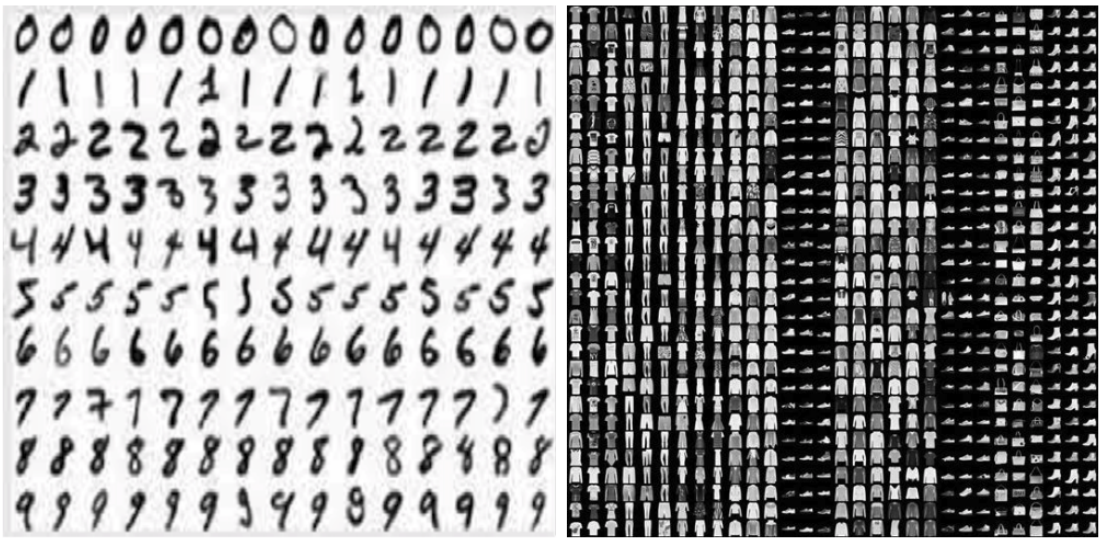

Figure 1.5: Digit MNIST and Fashion MNIST compared.

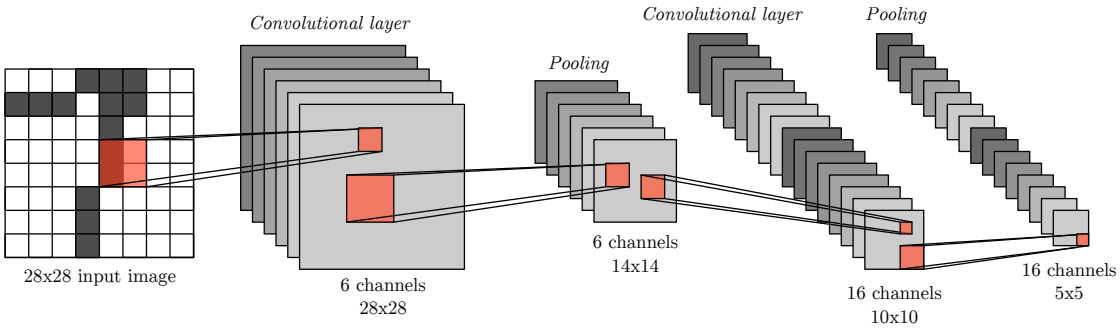

Figure 1.6: High dimensional convolutions requires a set of filters for each output channel of a certain layer. In every set, weights change for each input channel. Feature maps size depends both on the *kernel* width and the *stride*, which indicates the amount of pixels the filter is shifted after each convolution. The picture shows convolutional layers of a LeNet [33].

Possible transformations are:

- A nonlinear activation function. The most used is the rectified linear unit (ReLU), but there exist other similar versions. Sigmoid and the Hyperbolic Tangent were also employed for this purpose in the past.

- Pooling is used for reducing the dimensionality of the activation feature map. This makes computation lighter and the network more robust to small shifts and distortions.

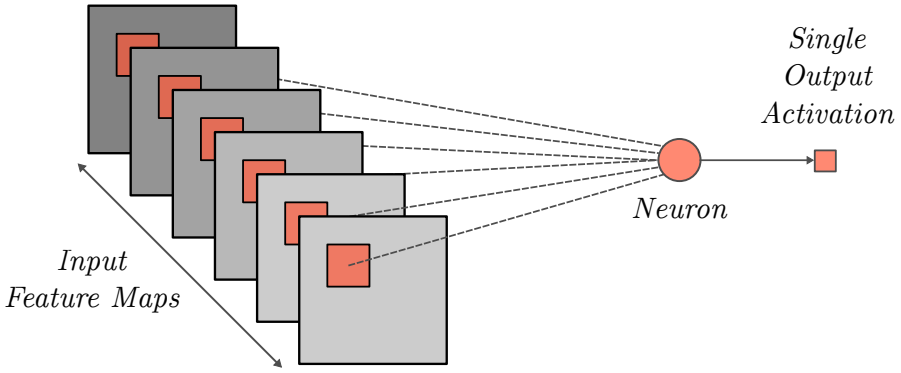

Figure 1.7: In this detail of a high dimensional convolution a set of 6 different kernels is needed for the computation of each output channel.

- Activations may be normalized for achieving zero means and a unit standard deviation to speed up training and improve accuracy.

After propagating inputs through the network, the predictions must be checked with true labels. For this purpose, several loss-functions can be employed to represent how far the results are from the real ones. Then weights should be updated for minimizing this quantity.
The most used algorithm to compute new values for the weights is still a modified version of the stochastic gradient descent used in the 60s.
This whole process is known as *back-propagation*, which uses the *chain rule* for the gradient computation of each weight, from the output back trough the data-path. Starting from the forward pass formula

$$y_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \tag{1.2}$$

where:

- y_j^l is the output of a certain **j** neuron in a **l** layer.
- w_{jk}^l is the weight **k** out of m, where m is obtained by the kernel size squared.
- a_k^{l-1} is the output activation of the previous layer.
- b_k^l is the bias, unique for every neuron.

The chain rule can be written as

$$\frac{\delta C}{\delta w_{jk}^l} = \frac{\delta C}{\delta y_j^l} \frac{\delta y_j^l}{\delta w_{jk}^l} \tag{1.3}$$

C represents the loss. In this way, it is possible to link the cost function with the variation of the weights. Kernels will then be updated through an optimizer. Convergence to the cost's minimum is ensured and the set of weights that produced the most accurate result will be stored in the network structure.
It must be noticed that gradients are not directly subtracted to the weight's value, but they are tuned through a learning rate (α), a hyperparameter used to speed up the training and to increase the sharpness in reaching the minimum.

$$w' = w - \alpha \frac{\delta C}{\delta w} \tag{1.4}$$

The forward-pass is performed for every set of inputs, called *batches* or *mini-batches*, from the training set. After this first step gradients are computed, evaluated and weights will be hence updated. The whole data-set is processed as many time as the number of epochs specified by the programmer.
One of the problems ANN designers must face is the over-fitting of the training data. For prevention, a common technique is data augmentation; it consists of slightly modifying input data to increase network capacity to correctly label inputs in a real-world scenario. A test must be performed without computing the gradients to check whether the training is incurring into over-fitting. The test accuracy is then compared with training one; if values are not coherent, the algorithm can be interrupted.

Chapter 2

Accelerating Deep Learning algorithm

2.1 Why specific hardware?

Machine learning was confined in the research field for many years due to the computation and storage complexity of networks. Then, improvements on technology allowed the rising of new studies focused on the possibility to implement artificial intelligence in several and heterogeneous platforms.
For understanding how complex is machine learning elaboration, Table 2.1 shows the characteristics of most known networks designed for ImageNet classification.
Typical CPUs perform 10 to 250 GFLOPS with less than 1 GOP/J of efficiency

DNN	Year	# Parameters	# Operations	Top-1 Accuracy
AlexNet [25]	2012	60M	1.4G	61.0%
VGG19 [34]	2014	144M	39G	74.5%
ResNet152 [35]	2016	57M	22.6G	79.3%
MobileNet [36]	2017	4.2M	1.1G	70.6%
ShuffleNet [37]	2017	2.36M	0.27G	67.6%

Table 2.1: Examples of modern deep neural network complexity [38].

(e.g. Intel i9 9900K reaches 236 GFLOPS).
Running classification for most complex networks will require some time with significant energy consumption. Performing training on CPU would be then prohibitive. This is the main reason why GPUs, which reach 14TFLOPS at a lower power budget, are widely used and supported by many frameworks such as Caffe[39],

TensorFlow[40] or Pytorch[41]. The Nvidia Pascal architecture can even feature two 16-bit *brain floating-point* operations on a single-precision core as an otimization for DNNs.
FPGA based platforms are also possible for facing challenges in performance and power budget flexibility. But specific design requires heavier work than developing a CNN model with existing deep learning frameworks[38].
Examples of systems with a dedicated hardware for DL are the Facebook's Big Basin custom server[42] or the Google TPU based Cloud; there are even mobile platforms such as Nvidia Tegra and Samsung Exynos that support machine learning.
Unfortunately it is often required to design architectures according to specific needs, trying to minimize as much as possible non-recurring engineering costs. Specifications strongly depend on targets, and safety-critical contexts have to be taken into account. For example, for autonomous driving cars detecting a red light is fundamental, while the energy budget can be less relevant. Different is the situation in which AI must help tiny drones to avoid obstacles. In this case long-lasting life of the supply batteries is preferred, while for detecting impediments is enough a less accurate DNN elaboration [43].
Hence a broad solutions space is needed for fulfilling the possible needs. Fortunately, along with the classical approaches, machine learning allows brand new schemes.
Frameworks provide inference in single or double-precision floating-point. Back-propagation algorithm can be modified to train the wieghts in such a way they will result aware of the final architecture.
Intel [44], Xilinx [45] and others are providing new open source frameworks for supporting hardware-level design constantly updated to the state-of-the-art. This allows a fast analysis of the newest approaches and it contributes to create a standard way of designing neural networks for embedded systems.
In this work, Distiller by Nervana was chosen as support of analysis.

2.2 Hardware solutions for DNNs

This section will provide a fast overview of classical approaches for efficiently mapping neural networks in hardware, with a closer focus on specific structures of systolic arrays.
The kernel operation of DNN processing is the multiply and accumulate operation (MAC). Within the hardware perspective, such computation can be performed with high parallelism, exploiting both *temporal* and *spatial* architectures.
The order of operations in deep-learning algorithm can be modified for adapting to both different types of target platforms:

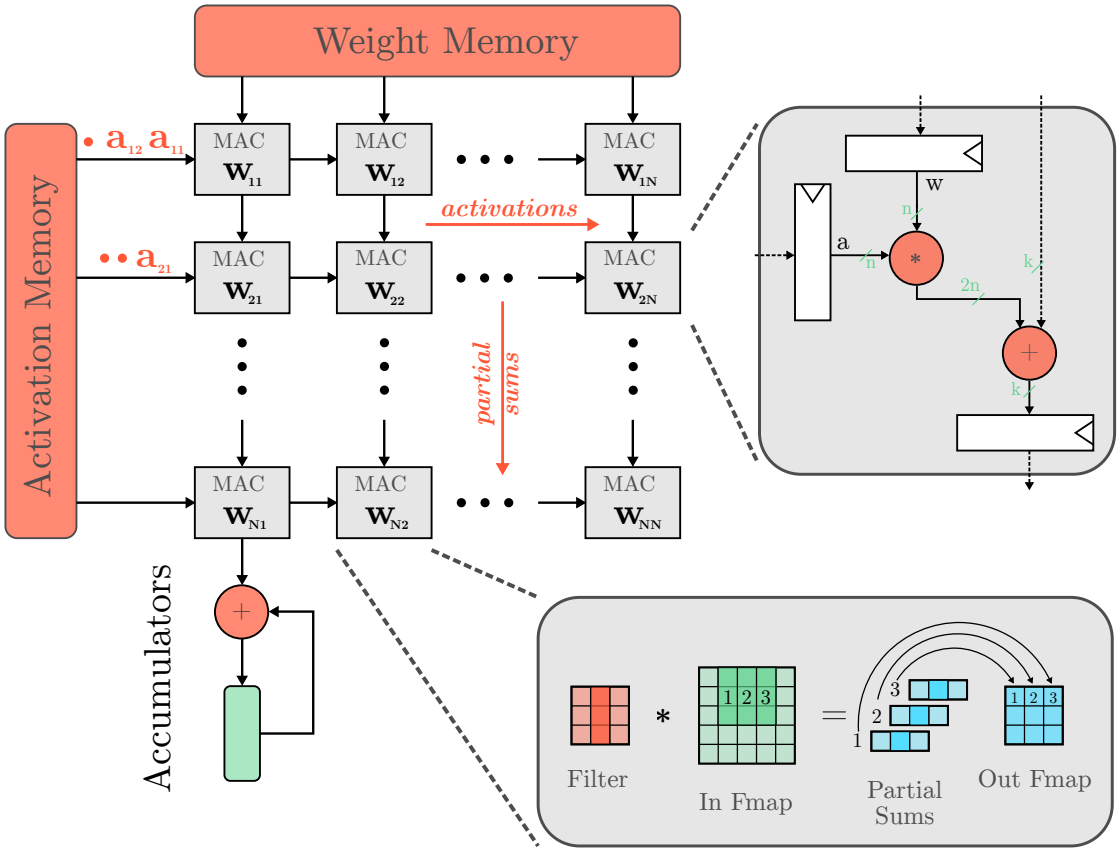

Figure 2.1: Systolic array is used for mapping network layers. In the bottom right there is an example showing computations performed in the second coloumn when kernel dimension is 3 and input feature map is 5x5 pixels[46].

- by reducing the number of multiplications it is possible to increase the throughput in temporal architectures like CPUs and GPUs.
- by introducing *data reuse* can be the key to energy-efficient spatial architectures (e.g. systolic structures in Figure 2.1).

The bottleneck in NNs is the memory bandwidth. Three different input must be fetched for each MAC operation, then the result must be stored back, as shown in Figure 2.2. For reducing the energy required in memory interactions, data must be reused as much as possible.

The simplest approach comes from the fact that, within the same channel of the same layer, the output feature-map is always the combination of the same input-map with the kernel filter. Figure 2.3 shows this and others possible solutions for avoiding memory access at the algorithmic level.

For exploiting data reuse from the hardware side, it's possible to implement a

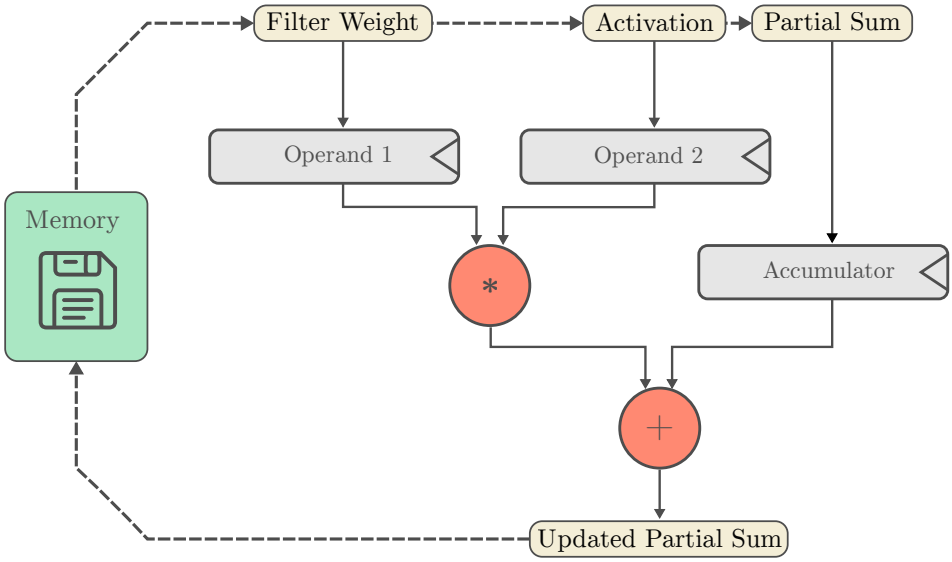

Figure 2.2: Data exchange during a MAC operation.

hierarchical memory system with buffers and local memories included in each arithmetic circuit. An ALU linked with a local memory of 0.5 to 1 kB is referred to as a *processing engine*. Several PE connected will create a systolic array. Common ways to reuse information are:

- keep the same input feature-map stored and apply the different channel filters to it
- keep in local memory the filters and apply them to batches of input data.

It is also possible to introduce a *global buffer* connected to all the PE that works as a memory and as an interconnection. Distributing the available chip area between local memory and buffer can provide several design solutions. Policies can be resumed into 4 categories.

- *Weight Stationary* approach minimizes energy consumption required for reading weights, storing them in the local register file. Activations are fetched from global buffer and output partial sums are accumulated letting them propagate through adjacent PEs.
- *Output Stationary* approach stores partial sums in local memory, streams input feature-map, and broadcasts weights from the buffer to PEs.
- *No Local Reuse* reallocates area used for local RF to global buffer to increase its capacity.

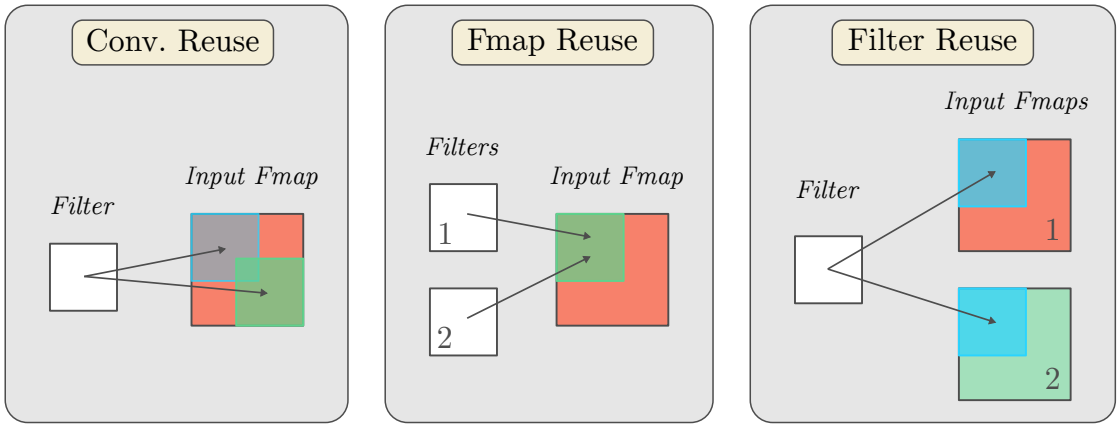

Figure 2.3: Common approaches for data reuse [1]. *Convolutional Reuse* is possible only for CONV layers with sliding windows: if local memory is enough both activations and filter can be reused for computing a single output channel. *Feature-map Reuse* can be exploited in all type of layers. The same happens for *Filter Reuse* as long as multiple input feature maps are available (e.g., batch size is greater than 1).

- *Row Stationary* approach maximizes the reuse and accumulation at the local memory level for all types of data. Kernel's rows are kept stationary inside RF, activations are streamed between PEs and the partial sums are accumulated inside the elaboration unit, then are passed to neighbor processing elements.

Hardware implementation and how layer are mapped depends on the policy chosen.
From the solutions shown above, RS is more relevant since it allows the design of *ordered structures* where DNN's layers can be automatically mapped by a (software) optimization compiler[1].

2.3 Sparsity

DNN's kernels can be more generally referred to as *tensors*. A tensor is by definition a mathematical object represented by an array of components that are functions of the coordinates of a space.
The concept of **sparsity** indicates how many entries of such structures are zero; a tensor without any null value is defined as *dense*.
Working with sparse kernel requires lower computational effort than with dense ones, and methodologies for increasing such property inside DNNs during training are nowadays under study.
Considering it from the software level, more weights are zero fewer multiplications

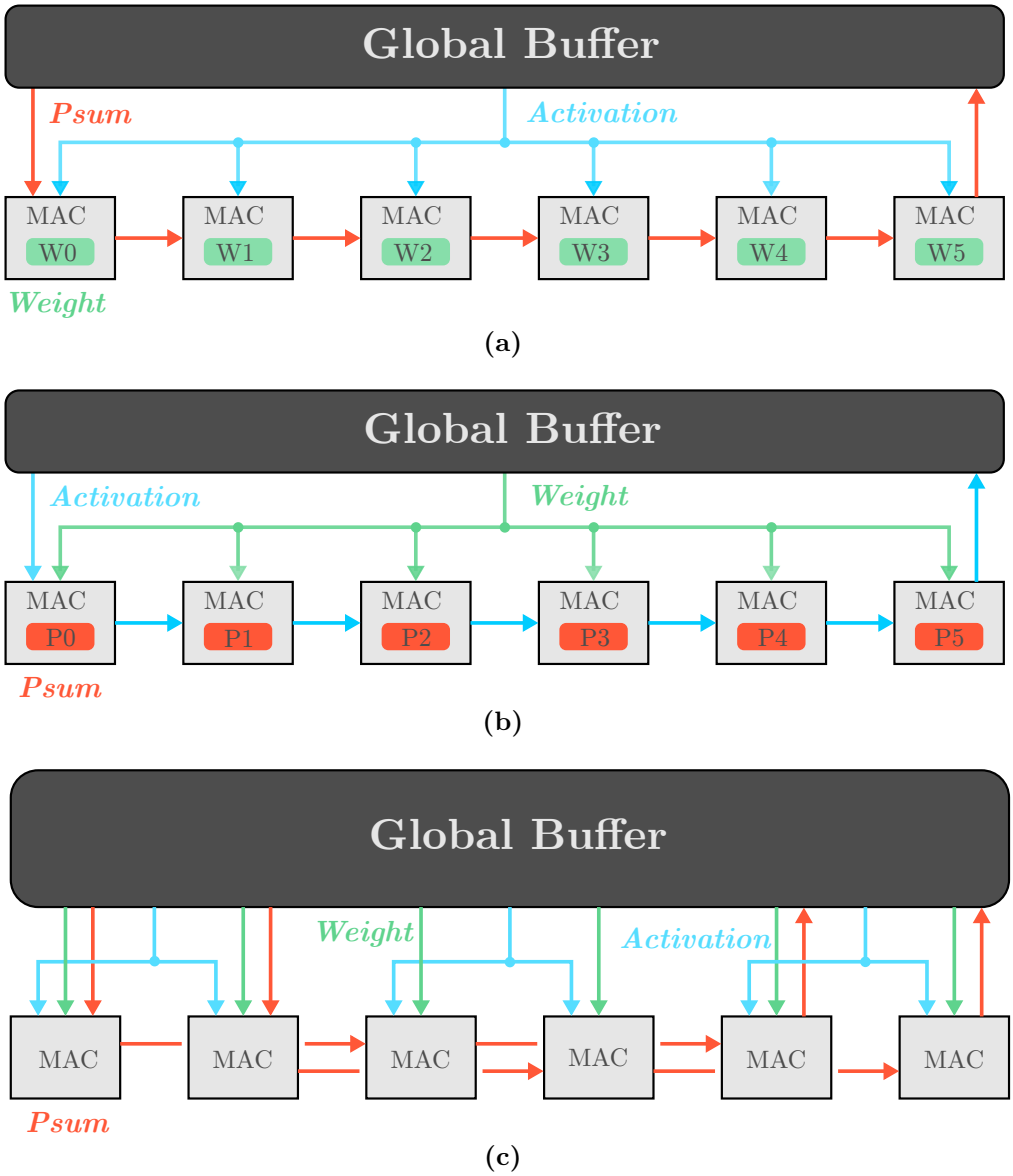

Figure 2.4: Different types of dataflows according to the chosen policy for designing the hierarchy of memories[1]. **(a)** Weight stationary. **(b)** Output Stationary. **(c)** No Local Reuse.

and additions would be required for completing the task. This can be translated into lower latency and less overall energy consumption. If we consider the hardware side, the intensive use of memories can be also restricted, opening new possible challenges in mobile environments.
It's hence manageable to obtain a compressed version of a model that acts exactly

as the original one but bringing more advantages in terms of performance and energy. These perks come at the price of more difficult software and hardware co-design, which becomes often application-specific.
Sparsity can be measured by the following formula.

$$\|S\|_0 = \sum_{i=0}^{N} |x_i|^0 \frac{1}{N} \tag{2.1}$$

where x_i is the i-th element, out of a total N, in the tensor. Then density becomes:

$$D = 1 - S \tag{2.2}$$

2.3.1 Pruning

A common methodology for achieving higher weights' sparsity is *pruning* [47] [48] [49] [50]. It consists of the application of binary flag to kernel's elements to decide which one to force to zero. This process, performed during training, lets the network adapt over the lack of some connections between neurons. A single or entire sets of close weights can be pruned.
Link trimming can be definitive or in some cases temporary to let the optimization algorithm find the best trade-off between sparsity and accuracy, and learn which connections are more important over the others.
Pruning can be applied both to weights and bias, but major advantages come with working with the first ones.
Clearly, deleting too many links will result in a loss of accuracy, while a less aggressive approach brings some extra benefits: it reduces redundancy and avoids over-fitting. The procedure may be also performed on an already-trained network, recent works show how it is possible to obtain the same accuracy halving the number of weights without any fine-tuning afterwards [51]. However, iteratively pruning connections during the training may lead to higher sparsity without any significant accuracy loss, furthermore, fine-tuning criteria and scheduling of the pruning operation make the network itself learn which connections are more significant. At every iteration, more weights are set to zero and the algorithm stops when either a target sparsity level is reached or accuracy decreases under a certain unacceptable value for the target application.
How connections should be deleted is chosen through a criteria; the most simple sets to zero all weights below a certain threshold; this procedure is known as *magnitude pruner*. It is possible to increase weights threshold more and more at each step

achieving a higher sparsity.

$$w_i = \begin{cases} w_i & \text{if } |w_i| > \lambda \\ 0 & \text{if } |w_i| < \lambda \end{cases}$$

Moreover, it's possible to choose a threshold λ for each layer that will represent the output sensitivity from each different level.

But since weights have different average absolute-values along with the layers, it

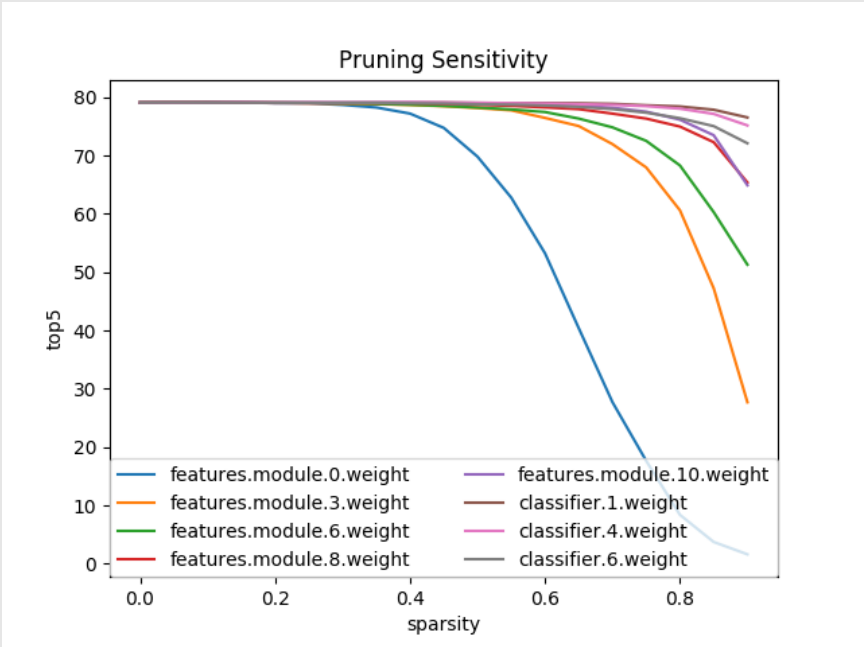

Image imported from pruning documentation of Distiller

Figure 2.5: AlexNet top5 error plot with the increase of sparsity level through layers[44].

requires some effort to find a single threshold for everyone of them.

To simplify and increase efficiency it's possible to exploit the common distribution of the kernel's values using the standard deviation as a sort of normalizing factor between the different tensors. The threshold is set multiplying σ to a sensitivity s value, known by performing the analysis with empirical methods as it was done in Figure 2.5 on AlexNet.

$$\lambda = \sigma * s \tag{2.3}$$

Sensitivity can be computed one time before pruning or can be updated at every training step. Other pruning approaches are the "Splicing Pruner" [52] and the

"Automated Gradual Pruner" [53], that won't be here analyzed in detail.
Pruning shows how networks are usually overparameterized. It was presented here for complete picture of modern newtwork design approaches, but it is not directly related with this thesis work.

2.3.2 Quantization

Frameworks implement deep learning with data in standard 32-bit floating-point format requiring high bandwidth and computational effort. Implementing a mobile accelerator in floating-point would be prohibitive. Fortunately, it has been extensively demonstrated that both weights and activations can be represented in a lossy 8-bit integer without effecting a significant accuracy drop.
Both inference and training can be implemented using integers but quantization is here studied just for the classification process.
Problems arise in the representation of wide distributions of data. For accurately

	Energy Saving INT8 vs FP32	*Area Saving* INT8 vs FP32
Add	30x	116x
Multiply	18.5x	27x

Table 2.2: Comparison between 8-bit Integer and 32-bit Floating Point convolutional operations[54].

handling these issues, scale factors must be associated with each tensor. These values are usually represented in floating-point 32-bit, even though there are ways to proficiently approximate them [55].
Hardware design with integers number must be carefully sized for avoiding overflows. Therefore, accumulators must be implemented with higher bit-widths respect to activations and weights. Neurons' output is the result of $c * k^2$ additions, where c is the number of channels and k is the kernel dimension. Hence partial sum parallelism, m, is given by the following formula:

$$m = 2n + \log_2(ck^2) \tag{2.4}$$

n represents the parallelism of the weights and the activations.
Bias and weights can be processed offline, but activations distribution unfortunately changes run-time, so they must be approached differently: range limits can be collected offline by obtaining statistics during the training or running few calibration batches on the trained floating-point model. This method, if statistics aren't exhaustive, won't adapt to possible distribution of new data and the model accuracy will decrease in real-world application (Figure 2.6). A possible solution is

to compute run-time the actual range of feature-maps on each layer at the price of extra resources.

There are many techniques to mitigate the effects of quantization. One can re-train

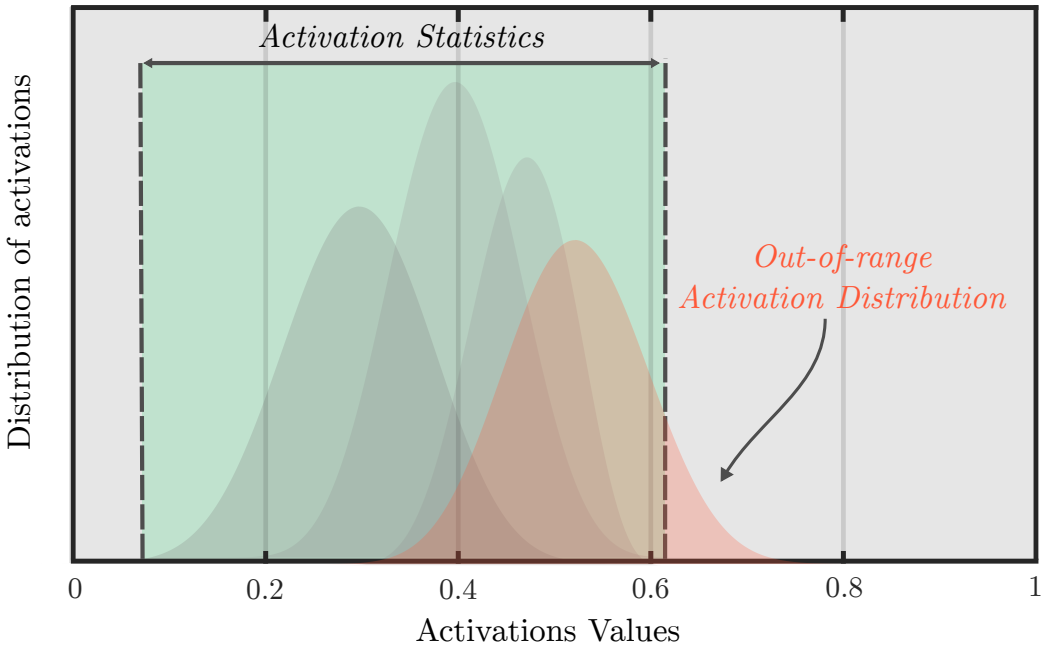

Figure 2.6: Example of activation distribution exceeding the statistic range computed offline, during real-world inference.

the network and fine-tune the result, or, in another way, improve the resolution of value representation by *clipping* the data close to the minimum or the maximum, as shown in Figure 2.7. An efficient clipping strategy is using an average of the actual min/max values, or, for example, statistically clip the range to preserve more information.

Quantization-Aware training remains, as when performing pruning, the best way for achieving almost the same accuracy of reference FP networks. It comes with a heavier computational burden just in the training algorithm.

Functions used for quantization are discrete-valued, their derivative would then be zero almost everywhere and including it in the back-propagation will degrade the learning process. A *straight-through estimator* approximation is used for passing the gradient through quantization function as it is [56].

Algorithms used in this thesis work lay on a Range-Based Linear Quantization: linear means that float values are quantized through multiplication with a numeric constant (the scale factor), range-based means that scale factor is obtained by looking at the range of original tensor values. There exist two methods to go:

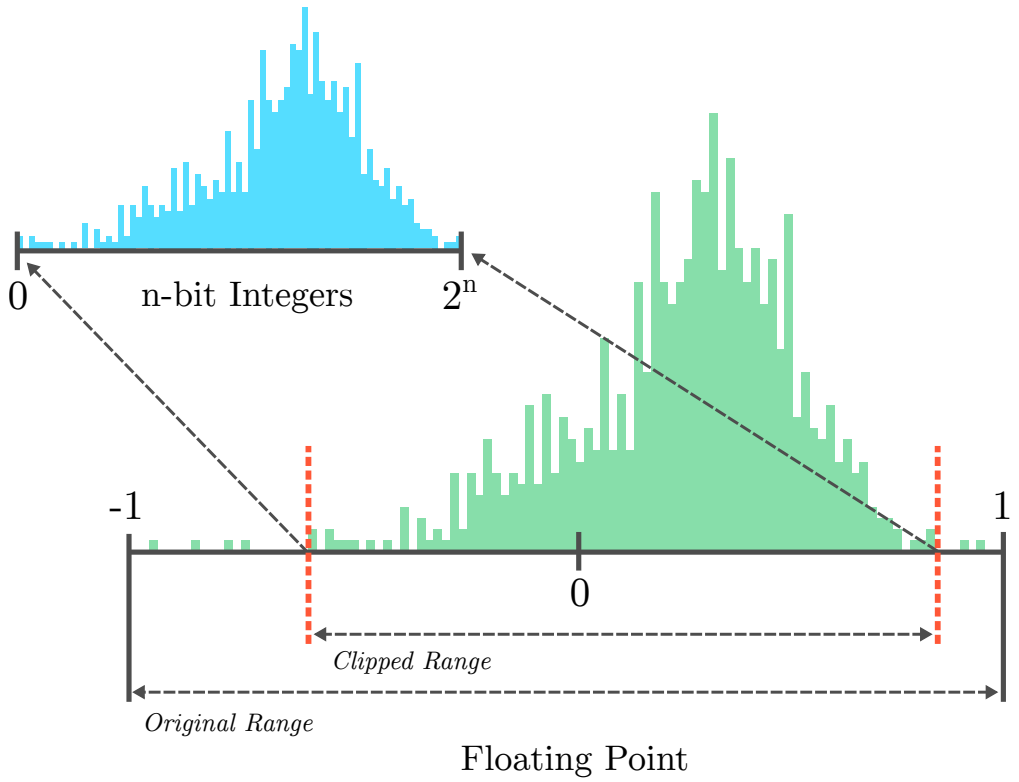

Figure 2.7: Example of *unsigned range* mapping with the clipping of outer data.

asymmetric and symmetric. In *asymmetric mode*, the minimum and the maximum of the float range are mapped to *zero* and $2^n - 1$, where n is the number of bits used for the final representation. In this case, *zero is mapped with an integer in the new range*. The formula for quantizing tensors asymmetrically is

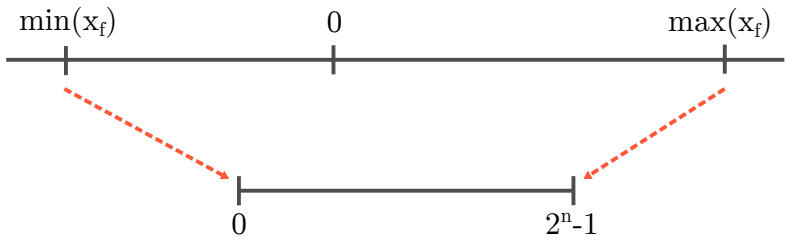

Figure 2.8: Quntization with unsigned asymmetric range mapping.

$$a_q = round\left(\left(a_f - min(a_f)\right)\frac{2^n - 1}{max(a_f) - min(a_f)}\right) \tag{2.5}$$

The scale factor can be indicated simply with s_f and the zero point as z_p.

$$s_{fa} = \frac{2^n - 1}{max(a_f) - min(a_f)} \tag{2.6}$$

$$z_{pa} = min(a_f) s_{fa} \tag{2.7}$$

Transformations applied to the data must be propagated through the network as well. Using a *light* notation is possible to write

$$y_f = \sum a_f w_f + b_f = \sum \frac{a_q + z_{pa}}{s_{fa}} \frac{w_q + z_{pw}}{s_{fw}} + \frac{b_q + z_{pb}}{s_{fb}} = \tag{2.8}$$

$$= \frac{1}{s_{fa} s_{fw}} \left(\sum (a_q + z_{pa})(w_q + z_{pw}) + \frac{s_{fa} s_{fw}}{s_{fb}} (b_q + z_{pb}) \right)$$

Therefore the quantized output would be

$$y_q = round \left(\frac{1}{s_{fa} s_{fw}} \left(\sum (a_q + z_{pa})(w_q + z_{pw}) + \frac{s_{fa} s_{fw}}{s_{fb}} (b_q + z_{pb}) \right) \right) \tag{2.9}$$

By means of some arithmetic transformations, it's possible to come up with a standard version of multiply and accumulate algorithm that can be executed by architectures described before.
In *symmetric mode*, the zero-point is always mapped with zero. For mapping the range then it is chosen the maximum absolute value x_f of the floating-point tensor X_f. As already done for the previous case, let's derive the output formula.

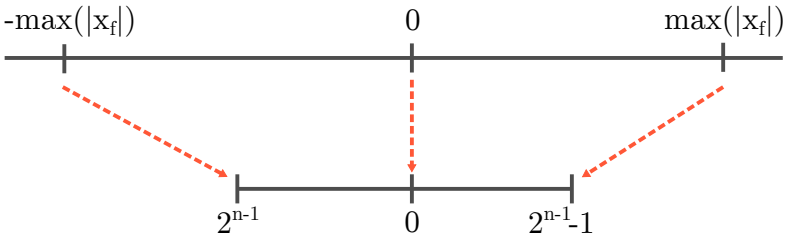

Figure 2.9: Quntization with symmetric range mapping.

$$a_q = round \left(\frac{2^{n-1} - 1}{max(|X_f|)} x_f \right) = round(s_{fa} x_f) \tag{2.10}$$

$$y_f = \sum \frac{a_q}{s_{fa}} \frac{w_q}{s_{fw}} + \frac{b_q}{s_{fb}} = \frac{1}{s_{fa} s_{fw}} \left(\sum a_q w_q + \frac{s_{fa} s_{fw}}{s_{fb}} (b_q) \right) \tag{2.11}$$

The quantized velue is then obtained by rounding

$$y_q = round\left(\frac{1}{s_{fa}s_{fw}}\left(\sum a_q w_q + \frac{s_{fa}s_{fw}}{s_{fb}}(b_q)\right)\right) \tag{2.12}$$

Many frameworks such as TensorFlow, NVIDIA TensorST and Intel DNNL use a restricted range that doesn't include the most negative value -2^{n-1} like the Equation 2.10.
If the two methods have to be compared, the symmetric mode is easier to implement in hardware but the quantized range may be less utilized resulting in a less accurate value mapping. For example, after ReLUs tensors are entirely positive, thus the negative range is never used and half of the available bins are wasted. On the other hand, asymmetric mode requires extra logic for zero-point handling that affects latency, power and/or area of the resulting design.

Chapter 3

Related Works

This chapter will introduce state-of-the-art related work, that provided fundamentals and inspiration for this thesis. Researches addressed deep neural networks' resilience and capability to positively adapt to some type of correction; the goal was improving AI in terms of energy consumption and portability. A brief description of research paper will be here presented, highlighting contributions to this work.

- Deep Compression [57] and Clip-Q [7] focus on the possibility to reduce redundancy inside networks and compress their size mainly at the software level
- AxTrain [6], CANN [58] try to implement approximate hardware in DL acceleration
- ALWANN [5] provided an approach to populate Pareto optimum curve when a big space solution has to be explored

All images and data shown are taken from the respective work.

3.1 Deep Compression

"Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding"[57] proposes what was a novel approach to deep neural network *distillation*. The target is to reduce as much as possible the memory size of network parameters. For this purpose three steps were manly elaborate on the paper:

- *pruning* is used for removing the connections that are not considered important for the classification [51]

- *quantization* then is applied for clustering weights and reduce the overall size of the kernels

- lastly for further compress the network, the *Huffman code* is used for addressing parameters

The main focus is set on how to store weights in memory. Connectivity of network is at a first step learned normally, then *magnitude pruner* is used for removing all weights below a certain threshold. Finally, retraining is performed for fine-tuning the weights of remaining sparse connections. Non-zero values are stored in compressed sparse row or column, recording indexes difference instead of absolute position. This allows using 8-bit addresses for convolutional layers and 5-bit ones for fully connected.
Quantization is then applied to find centroid of clusters that can collect as many weight as possible. Each bin is then fine-tuned uploading the centroid with the sum of the gradients of the weights related to that specific bin. An example (Figure 3.1) of a 4x4 kernel tensor is provided.
The compression rate reached in this way can be expressed as

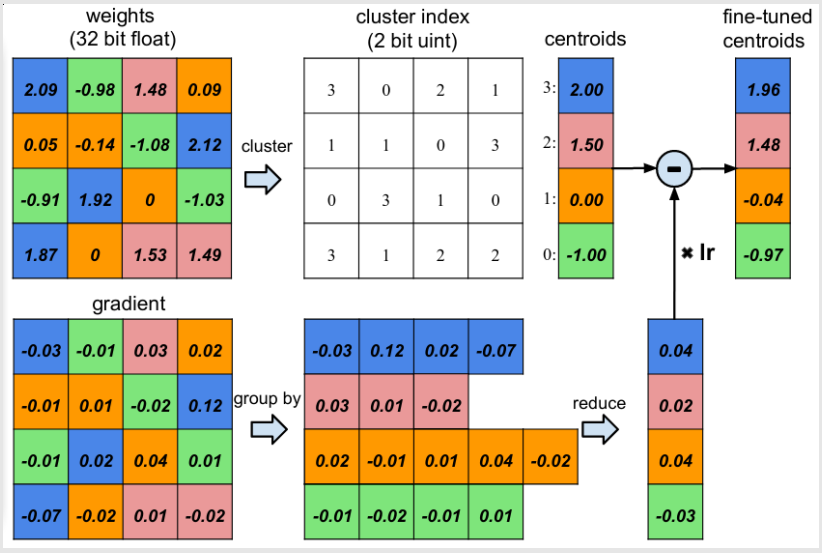

Figure 3.1: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom)[57].

$$r = \frac{nb}{n\log_2(k) + kb} \tag{3.1}$$

Where n is the number of original connection and b the number of bits required to represent each one of them. nb is the size of the reference network. k is the number of effective number of centroids, which take kb bits of memory; $n\log_2(k)$ is then required for storing the addresses.
To minimize the sum of the distance between weights and their representation, the paper shows that cluster centroids should be initialized linearly spacing them between the minimum and the maximum value of the original tensor. This will result in a more versatile and scattered distribution.
As the last step, weights are coded using Huffman approach, which is out of scope for this work.
Results achieved with this methodology are interesting, since FC layers of VGG-16 were reduced by up to 98.9% of their original memory size (Table 3.1).
Looking at the results obtained by Han et al. is clear that deep neural networks

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	27 KB	40x
LeNet-5	0.8%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	44 KB	39x
Alex-Net	42.78%	19.73%	240 MB	
Alex-Net Compressed	42.78%	19.70%	6.9 MB	35x
VGG-16	31.50 %	11.32 %	552 MB	
VGG-16 Compressed	31.17 %	10.91 %	11.3 MB	49x

Table 3.1: The *deep compression* pipeline can save 35× to 49× parameter storage with no loss of accuracy.

are typically over-parametrized. Their technique targets different mobile platforms, but other ways can be explored if design space is restricted to ASICs.

3.2 CLIP-Q

Another work that more deeply tries to reduce the degree of parameter redundancy in deep neural networks is *"CLIP-Q: Deep Network Compression Learning by In-Parallel Pruning-Quantization"* [7].
Novelty represented by this work are three:

- *pruning and quantization* are here included in the same framework
- decision for both connection trimming and quantization aren't permanent but they adapt overtime in the training as network structure changes

- the two techniques are performed in parallel in such a way one adjusts to the other, achieving better result in terms of accuracy, loss and sparsity.

The paper proposes a training which allows the network to learn not only to perform a specific task but also it's own shape progressively. Connections may be removed and then restored, bit range may decrease and then adapt again to new structures, all these for minimizing both the loss and the density.
The training algorithm consists of three steps: clipping, partitioning and quantizing. During the whole process, a back-up of the full-precision model is updated during back-propagation, while the distilled one is used only for forward-pass. Once learning is completed only the compressed network is saved and used for inference. The structure of the algorithm is explained in Figure 3.2.
Clipping step is performed by setting the percentage p of weights (between 0 and 1)

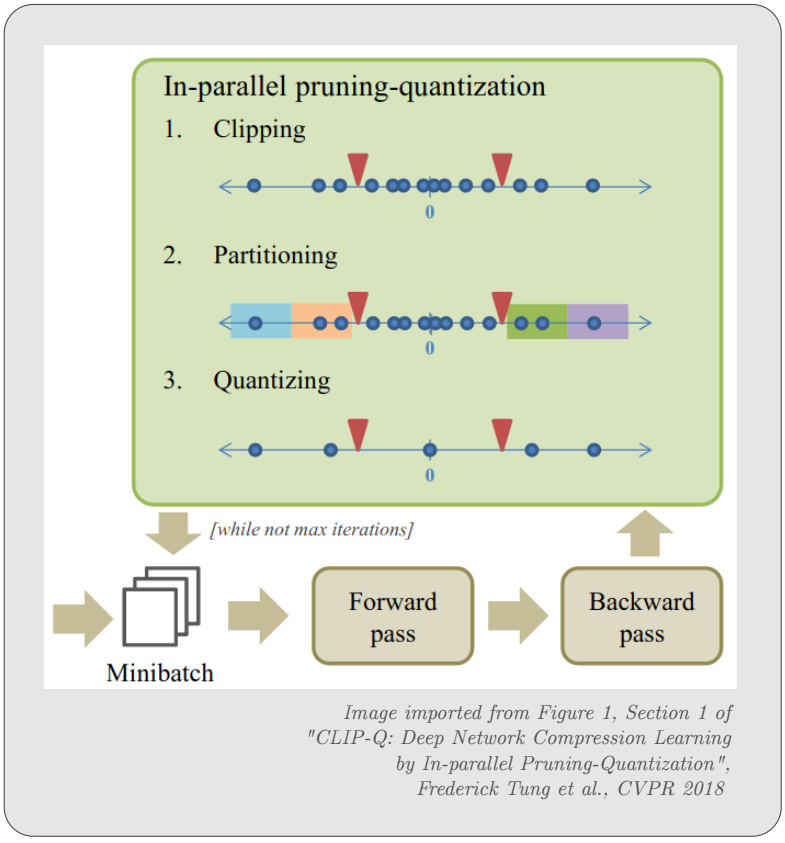

Figure 3.2: Overview of CLIP-Q algorithm. It combines weight pruning and quantization in a single learning step. The pruning-quantization adapts overtime with the changing network[7].

that will contribute as zeros in the forward pass of a minibatch of data. According

to this target, two thresholds are set, one for negative and one for positive range. It is important to notice that weights are not set to zero, but prediction, and then gradient and loss, are computed *as they were zero*. At the next step, given the same hyperparameter p, new thresholds are set and more connections are pruned. During *partitioning* weights are divided into quantization intervals depending on a precision budget b. Linear partitioning is then performed mapping values into 2^b bins. First one will be addressed by 0, while last one with $2^b - 1$.
In the end, the quantized value, that substitues all the wieghts falling into a specific cluster, are computed by averaging the full-precision vlaues within the corresponding quantization interval. Quantization applies only in the forward-pass as already said for pruning.
The hyperparameters p and b determine the sparsity and the accuracy of the network. In the paper, they are set layer-wise through a Bayesian optimization, which acts on a black-box objective function defined as

$$min_\theta \epsilon(\theta) - \lambda c_i(\theta) \tag{3.2}$$

θ represents a couple of hyperparameters, ϵ the top-1 error and c_i is

$$c_i(\theta) = \frac{m_i - s_i(\theta)}{\sum_i m_i} \tag{3.3}$$

that express the *gain* we have for each layer i. m_i is the number of bits requires to store the weights of layer i in the original network, while $s_i(\theta)$ the bits used for the commpressed one.
λ indicates how much compression is important in the optimization process. This methodology provides a reduction of 51 times the AlexNet overall memory size, with a drop of 0.7% in the accuracy of ImageNet classification (from 57.9% to 57.2%).
The important contribution provided by this work is that it is possible to *shape* the network while learning the parameters.

3.3 Ax-Train

"AxTrain: Hardware-Oriented Neural Network Training for Approximate Inference"[6] proposes a way to increase the intrinsic error tolerance of DNNs. A more resilient network allows the application of approximate computing for energy-efficient computation.
Resilience is developed by pursuing a set of parameters in the network highly error-tolerant and incorporating the expected noise distribution of approximate hardware in the forward pass during the training phase. The technique is thus orthogonal, providing improvements for both the software and the hardware side

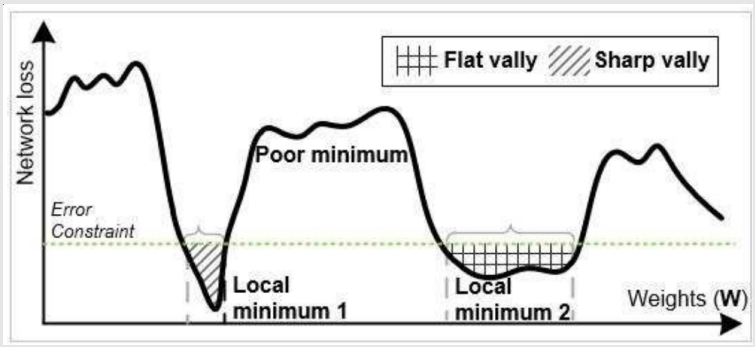

Image imported from Figure 1, Section 1 of "AxTrain: Hardware-Oriented Neural Network Training for Approximate Inference", Xin He et al., ISLPED 2018

Figure 3.3: Different type of minimum of the loss function[6].

to avoid or reduce as much as possible accuracy drop. Once again, acting during the training is fundamental to exploit awareness of the target hardware; moreover, AxTrain provides an active method that biases the network to a noise insensitive, accurate, minimum.
Considering the robustness of the network as

$$E_{tot} = E + \gamma S(w) \tag{3.4}$$

Where E is the original output error, γ is a preference factor that is used for fine-tuning the optimization process. Sensitivity $S(w)$ can be then defined for each layer l as:

$$S(w) = \sum_{k} \left(\sum_{ij} |w_{ij}| \left| \frac{\delta O_k}{\delta w_{ij}} \right| \right) \tag{3.5}$$

where the absolute value takes into account the worst-case scenario. The derivative indicates each output k (O_k) response concerning weight perturbation, while $|w_{ij}|$ represents the fact that noise is proportional to value's magnitude. $S(w)$ has to be taken into account during back-propagation as it is commonly done for the loss function.
For what concerns learning the noise distribution from inaccurate hardware, approximate multipliers and near to threshold voltage memory are considered. AxTrain models NTV induced flips in the forward pass as stochastic noise, while approximate arithmetic by performing computations with the software counterparts. Gradients in the backpass don't take into account these modifications, a straight to estimator approach is instead preferred.

With this methodology, up to 75.61% of errors are reduced for the most aggressive approximation, with close to half of the power dissipated.
This work underlines how different sets of parameters, that provide almost the same accuracy, can be significantly distant in terms of error tolerance (Figure 3.3).

3.4 CANN

Yet another approach to DNN with approximate hardware is represented by *"CANN: Curable Approximations for High-Performance Deep Neural Network Accelerators"*[58]. The paper points out several lacks of other previous approaches in the context of safety-critical applications, where it is still needed a high level of energy-efficiency. Accuracy drop in such scenarios is prohibitive, preventing the use of classical approximate arithmetics.
Even AxTrain approach won't be feasible in this case since additional constraints on the training process limit the learning capabilities of the network and will prevent to reach tolerable error in many applications.
For what concerns reconfigurable hardware design, the overhead introduced by multiplexers and the area taken by multiple architectural solutions may overcome IC constraints and prevent practicability.
The solution provided exploits the structure of the systolic array: at every PE an approximate result is transmitted to the next one along with a correction bit, that can be easily handled at the next step by a Wallace tree multiplier. Differences between the proposed design and classical ones are shown in Figure 3.4, along with analyzed characteristics in Table 3.2.
With this approach, critical paths of the architecture are significantly reduced and the clock frequency can be lowered for important energy savings. The power-delay product of the proposed hardware is close to half of the conventional one.
What is important here is not the solution provided, yet the problem itself. The

	Latency [ps]	Cell Area	Power [μW]
Accurate MAC (Merged)	1871.1	746	66.56
DAx MAC	1214.2	744	66.3
C&DAx MAC	1214.2	746	68.13
Accurate MAC (Conventional)	2470.9	889	62.73
Approx MAC (Truncating 3 LSB)	2274.2	822	61.14

Table 3.2: Hardware characteristics of different types of MAC units.

challenge is to commit an architecture with shorter paths, thus allowing energy savings, without any impact on the accuracy of the network.

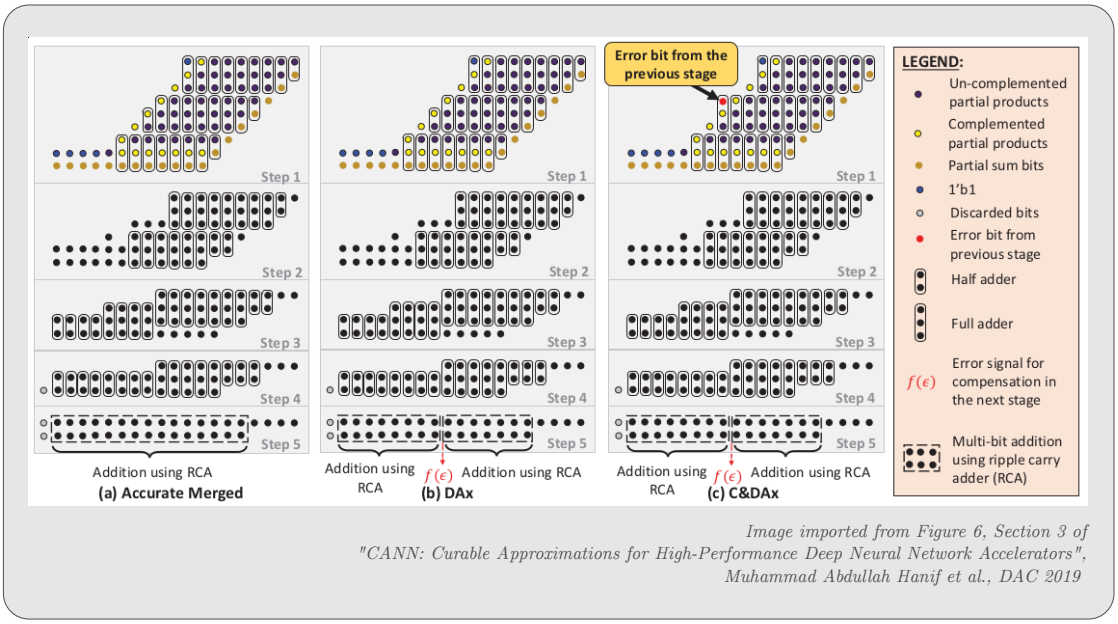

Image imported from Figure 6, Section 3 of "CANN: Curable Approximations for High-Performance Deep Neural Network Accelerators", Muhammad Abdullah Hanif et al., DAC 2019

Figure 3.4: Design of different types of MAC units based on Bough-Wooley multiplication algorithm and Wallace tree architecture. The multiplicand and the multiplier are assumed to be 8-bit wide and the partial sums are assumed to be 19-bit wide.(a) Accurate Merged MAC. (b) Deterministic Approximate (DAx) MAC. (c) Cure amd Deterministic Approximate (C&DAx) MAC[58].

3.5 ALWANN

"ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining"[5] proposes a way to avoid time-exhaustive retraining of networks designed with approximate arithmetics. They provided an approach that orthogonally optimizes the model, taking into account both the classification error and the energy consumption.
The paper proposes two contributions to the automatic design of hardware neural networks:

- the tuning of weights according to the properties of the approximate multiplier used in the design
- the automatic generation of a Pareto front of solutions through a multi-objective optimization NSGA-II algorithm

Approximate computing affects more or less aggressively the outcome of the network. Knowing a priori the structure and the value of weights, it is possible to mitigate these errors. Through a set of offline mapping functions, kernel's entries can be

tuned for minimizing the sum of absolute differences between the output of a specific approximate multiplier and the accurate one. In other words, functions are chosen such that

$$\forall w \in W : f_{M_i}(w) = \min_{w' \in W} \sum_{a \in I} |M_i(a, w') - a \cdot w| \tag{3.6}$$

where w is general weight of the kernel W, M_i is an approximate multiplier solution, and a is an input of the I set.
For what concerns the solution space exploration, the genetic algorithm finds specific network configurations, evaluates them in terms of accuracy and energy consumption, and then saves only the non-dominated ones.

Each solution found maps the layers of the network model into the IC's tiles.

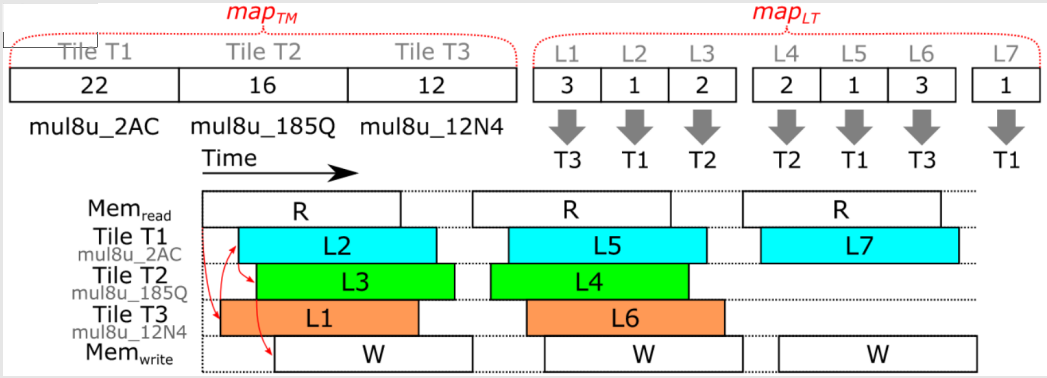

Image imported from Figure 6, Section 3 of "ALWANN: Automatic Layer-Wise Approximation ofDeep Neural Network Accelerators without Retraining", Vojtech Mrazek et al., ICCAD 2019

Figure 3.5: Example of an encodings of approximate neural network with 7 layers and 3 tiles. map_{TM} indicates which multiplier solutions have been assigned to each tile, while map_{LT} presents the mapping of layers into the different section of the integrated circuit. The timing diagram shows the execution of an inference according to network's dataflow[5].

Every tile employs a specific approximate multiplier from a library provided by the user. An example is the Figure 3.5. In this way, a set of valid configurations is automatically provided, and the designer can choose the one that fits the most for his target application.
With the ALWANN approach applied to ResNet-50 for CIFAR-10 dataset classification, they obtained 30% energy saving with less than 0.6% drop in the network accuracy.
A relevant aspect of the work is that the optimization algorithm provides not just a

single solution but a set of configurations that may be equally valid if the contexts of deployment changes.

Chapter 4

Bit-level masking of weights and software analysis of the effects on network accuracy

4.1 Masks: forcing kernels' bit to zero

As previously said, this work aims to provide and to analyze a new approach to reduce timing delays of PE in deep learning accelerators.
Timing is a crucial factor in most applications, from data centers to those in the edge of mobility. [1] Many techniques target the design of the datapath in order to speed up the execution, but, since network accuracy has a big impact on the final application, it is challenging to set up a new approach that allows better performance to be compliant with very low error rates.
Delay constraint is directly linked with the critical path timing: if it is not taken into account at design time, execution may incur in timing errors. Most of the classical technique, as pointed out in [46], result in an energy loss or a significant degradation of the network precision.
How can the critical path be shortened to avoid clock violations?
Let's consider a common PE for DNN, with 8-bits weights, 8-bits activations, and 32-bits partial sums. It's easy to detect that longest propagation happens through the multiplier and then the adder (Figure 4.1).
Fast arithmetic solutions and approximate designs have been widely exploited in many previous works [6][5][3].
The concept presented found inspiration from the possibility to analyze which weights of each layer-set trigger the slower paths during execution; if those values are then excluded from the inference process, it's possible to design an architecture

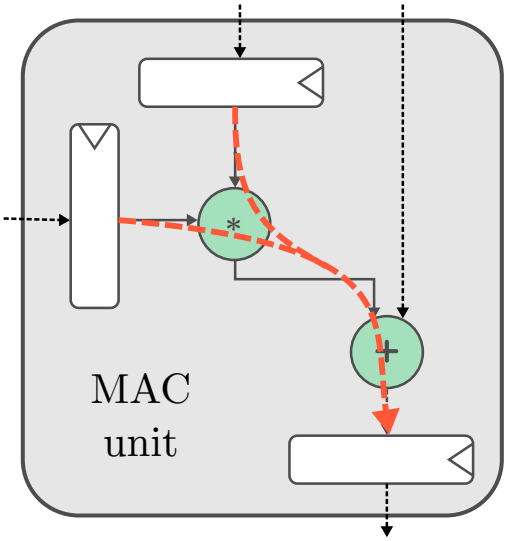

Figure 4.1: Critical paths in a MAC unit.

working at a faster frequency without incurring in any timing error.
By looking at the circuit, it is not trivial to detect which values activate exceding paths. The solution found does precisely the opposite process: it generates a design depending on which set of numbers is chosen to be *"free to use"* for performing the *multiplication*. It must be noticed that it's not essential to know the activation value to proceed in this direction: only the weight is needed.
In 8-bits tree multipliers, eight partial products must be generated and then summed. But if one of the i-th bit of the control-operand for the multiplication is known at design time to be never 1, the hardware required for performing the i-th addiction can be dropped, allowing faster and equally accurate hardware. Figure 4.2 pictures an example of how a mask changes the computation of products at hardware level.

The operation of forcing a specific bit to 0 is be called *masking*. An n-bits weight with k of its digit masked will require a hardware comparable to the one needed as if it had $(n-k)$-bits. The benefit comes from the fact that different masks can be applied layer-wise, setting different ranges of forbidden values. Weights' distribution will be modified (Figure 4.3 shows a possible outcome), but the oscillations can then be balanced and optimized.

4.2 Masking algorithms

Masking must be applied after the quantization of deep neural networks. A complete software analysis is needed to understand and inspect how the models will behave

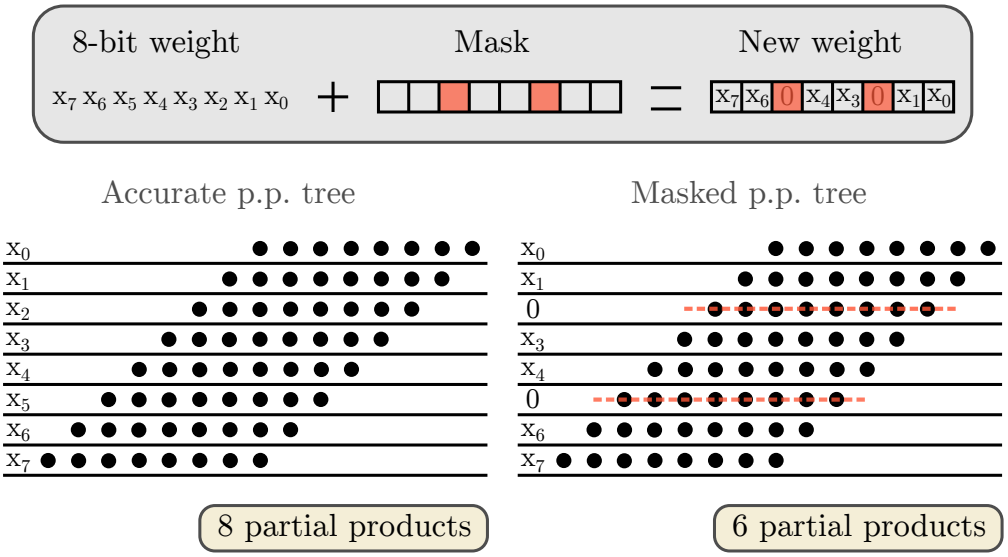

Figure 4.2: An example on how a possible mask changes the computation of a product at the hardware level.

once deployed on a real platform.
As a first approach, the technique was used on networks previously trained on full precision floating point data, for CIFAR-10 image recognition. Then an algorithm for mask-aware training was implemented.
Before going into the details of *when* to replace the weights, it was fundamental to understand *how* this operation should be done; parameter distribution, layer by layer, determines the overall performance of the artificial intelligence, hence reducing the impact of any modification on the network is fundamental.
It's possible to describe the replacement mathematically as

$$\forall w_i \in W_F \;\; \exists \;\; w_i' = f(w_i) \;\; | \;\; w_i' \notin W_F \tag{4.1}$$

where W_F indicates the complete set of forbidden values and w_i is the i-th weight of the network to be masked.
The distance between the two numbers is then

$$e = |w_i - f(w_i)| = |w_i - w_i'| \tag{4.2}$$

For minimizing e, a proper replacement function $f(\cdot)$ must be selected.
Another essential aspect that must be taken into account is the computational load that this operation requires. If an hardware-aware training is preferred, the masking algorithm must be executed N time during a whole learning process, where N is:

$$N = n_{epochs} \frac{n_{samples}}{b_s} \tag{4.3}$$

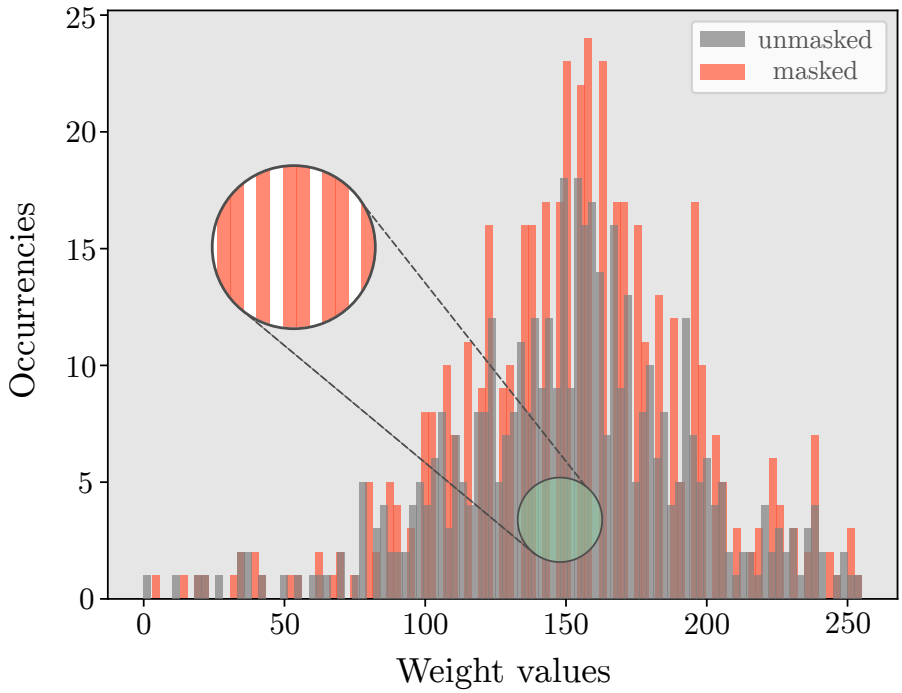

Figure 4.3: Weights' distribution before (in grey) and after masking (in red). The represented layer is the first convolutional of a *LeNet-5* [33] trained on CIFAR-10 and quantized asymmetrically. The applied mask sets to 0 the last *two least significant bits*. It is easy to notice that the red distribution has *"white bands"* where values are forbidden, and *spikes* (higher occurencies) for those that are allowed.

n_{epochs} is the total amount of epochs that must be performed, $n_{samples}$ refers to the number of samples in the training set and b_s is the batch size. A parallel approach is mandatory, that must also be compliant with the underlying framework and it can run both on CPUs and *on GPUs*.
The simplest implementation of the substitution is to apply a mask through logical *AND*. If, for example, the k-th bit of a value has to be forced to 0, a *bit-wise AND operation* must be performed with the "flipped" integer value 2^{k-1}.

$$t'_i = t_i \wedge \neg(\sum_{k=0}^{N} m_k 2^{k-1}) \tag{4.4}$$

t_i indicates a tensor of numbers, while t'_i its masked counterpart. m_k is a control binary value that tells if the k-th bit has to be masked (1) or not (0).
Unfortunately analysis on AND-masking revealed a very high drop in the accuracy of DNNs. Let's consider the case in which the value 16 is not allowed. The direct substitution would be 0 with an absolute distance of 16. With another approach, 15 or 17 can be chosen as a new weight, with a much lower error.

For every number that must be replaced, there is always a set of two allowed, one higher, and the other lower in value as shown in Figure 4.4. It is always preferred to choose the closest one, but a faster policy can also be used for lower computational load.

Three parallel substitution methods have been implemented:

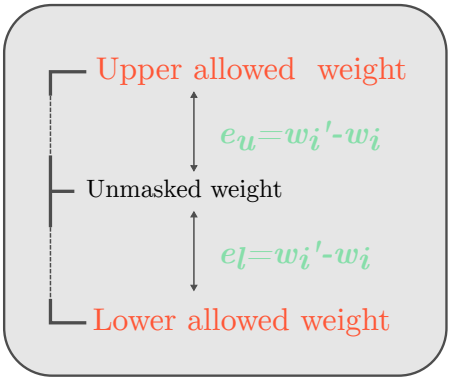

Figure 4.4: For every weight that must be replaced, there is always a set of two allowed numbers that are one higher and the other lower in value. Each one gives a different error distance e_u and e_l.

- Always "round" down (ARD): all the forbidden values are replaced with the closest smaller one.
- Always "round" up (ARU): all the forbidden values are replaced with the closest upper one. An exception is made for the cases that overflow the allowed dynamic. In such instances, the smaller is preferred.
- Always "round" to the closest (ARC): forbidden values are replaced with the one that provides the smaller error distance. Algorithm 2 presents the pseudo code of a possible implementation of this policy.

What happens if the *upper* and the *lower* distance are equal in ARC policy? To prevent bias towards one or the other direction, that may be very harmful to the network accuracy, the chosen number for the substitution is obtained through a random function. A complete version of the algorithm implemented in Python is aveilable in the Appendix.

As just said, the critical point while masking is to reduce as much as possible the bias produced by the replacement process. A masked weight w_i' is used for several multiplications before producing a single neuron output. The error distance that contributes in a convolution is then counted once for each input channel; if multiple parameters have a bias that tends in the same direction, feature maps can

Algorithm 2 Always round to the Closest (ARC) masking algorithm.

Input: tensor, bit_to_mask_list, dynamic, signed
Output:masked_tensor

```
mask=generateNegMask(bit_to_mask_list)
booleanTensor= int(bool(tensor & mask))    ▷ forbidden values are highlited
▷ generate upper bound tensor
up_tensor=tensor+booleanTensor    ▷ forbidden values are incremented by 1
while booleanTensor.sum() do
    booleanTensor= int(bool(up_tensor & mask))
    up_tensor+=booleanTensor
end while
▷ generate downer bound tensor
booleanTensor= int(bool(tensor & mask))
down_tensor=tensor-booleanTensor ▷ forbidden values are decremented by 1
while booleanTensor.sum() do
    booleanTensor= int(bool(down_tensor & mask))
    down_tensor-=booleanTensor
end while
▷ detecting overflows in upper bound tensor
MAX_INT=2**(dynamic-int(signed))-1
overflowTensor=int(up_tensor>MAX_INT)
▷ computing distances
diff_up = up_tensor - tensor
diff_down = tensor - down_tensor
▷ mapping the substitutions into Boolean tensors
toUpper = int(diff_up < diff_down)*(overflowTensor ⊕ 1)
toDowner = int(diff_down <= diff_up) | overflowTensor
masked_tensor = toUpper*up_tensor + tensor*(toUpper⊕1)
masked_tensor = toDowner*down_tensor + masked_tensor*(toDowner⊕1)
return masked_tensor
```

be significantly differet from the reference ones. Simulations with aggressive masks result in highly biased prediction (e.g., it doesn't matter the input, the network always prefers a particular class).
The optimal approach to this problem would be to characterize layer-wise every policy, then try to choose the one that balances more the output distribution of weights. For sake of simplicity and for reducing the computational load required for network processing, policies are chosen unifromely. Moreover, it may be more efficient, as it will be shown in Section 3, to improve the network comparing each layer output activations.

4.3 Masking approaches

As previously mentioned, there are mainly two possible approaches to process neural network models:

- The architecture can be fully trained using floating-point with common frameworks, and only after it is quantized and subjected to masking.
- The training algorithm can be modified to let the network adapt both to quantization and masking.

There exist also mixed solutions in the literature. Applying a mask to a floating-point model and retrain it with a hardware-aware algorithm can be one possibility. Inceasing the aggressivness of masking across the layers with the increasing number of epochs can be another solution. The proposed approach opens to many scenarios that can be explored depending on waht is the target.
In this work, only the post-training masking is entirely explored since it offers a fast solution space population and a prediction on how DNNs behave along with masks.
All the analysis done lay on the Nervana Distiller framework [44], which is built on top of Pytorch as shown in Figure 4.5. The open-source code of the tool has been modified for handling masking operations.

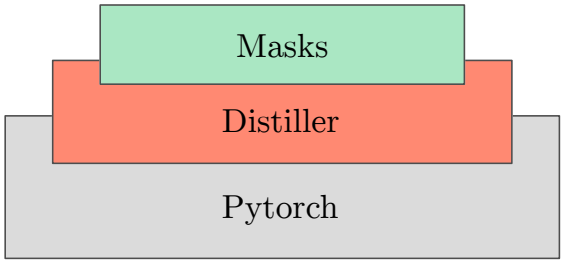

Figure 4.5: Mask framework is based on Distiller [44] and modifies functions and algorithms implementation. Distiller is itself based and settled into the most known Pytorch [41] framework for deep neural network design.

4.3.1 Mask-aware learning algorithm

As pointed out in the introduction chapters, training a network in such a way it will respond optimally to the target hardware is often the best way.
Building from scratch a specific framework for simulating the target hardware is time demanding and, in most cases, prohibitive. The trade-off is to modify existing

deep learning algorithm implementation to introduce awareness during the software execution.
Distiller from Nervana provides a coherent, hardware-aware overriding of the Pytorch framework to implement quantization in all the state-of-the-art proposals. The loss of resolution of values representation due to hardware restriction is simulated by applying the quantization of parameters before the forward-pass. Then they are de-quantized to avoid any conflict during the computation of the gradients by the Pytorch algorithm. The masking operation is performed between these two steps when tensor values are in a known range, and the behavior will be as close as possible to real hardware execution. A scheme representing the mask-aware training steps can be found in Figure 4.6

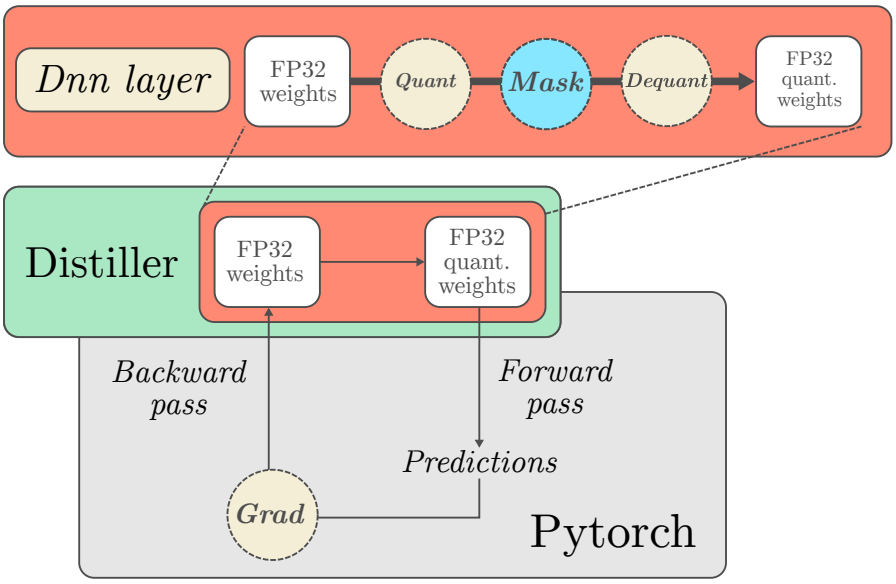

Figure 4.6: Graphical representation of mask-aware training.

4.3.2 Post-training masking algorithm

Post-training masking also happens after quantization. What changes is that after weights have been substituted with the allowed set of values, forward pass is executed keeping them, along with quantized versions of activations and bias, in an n-bit integers format. This approach ensures a more close to hardware behavior.

4.4 Experiments set up and accuracy issue

For testing mask effects on image recognition application, the CIFAR-10 dataset was chosen since it requires a tolerable computational effort with some challenges in achieving high accuracy. Network used as benchmark is the VGG11, in Figure 4.11, with batch normalization.
The first step in the analysis is to train the network for achieving state-of-the-art accuracy on the test set (Table 4.1). For this purpose, the training set must be augmented by means of random crop and random horizontal flips; the process took 300 epochs.
The optimizer used was the Stochastic Gradient Descent with a momentum equal to 0.9 and a weight decay of $5 * 10^{-4}$. A learning rate scheduler was also set for 0.1-decay every 90 epochs with a starting value of 0.1.

Network	**Accuracy**
VGG11	92.4%

Table 4.1: State-of-the-art network accuracy on 32 bit floating-point data.

After training, an evaluating loop was used for applying each possible mask to all the layers of the network uniformly to understand which ones were affecting more the performances. The asymmetric unsigned policy was chosen for the quantization. The results, in Figure 4.7 clearly showed that masking the least significant bits hits less the accuracy while processing MSBs leads to a sharp drop in the classification capability.
By comparing masks' performance with the one obtained through "aggressive" quantization (Figure 4.8), it was possible to notice that the new approach was giving worse results with comparable expected hardware performances. The gain obtained wasn't be enough to justify the effort of proposing a specific architecture.
Three methods were perfected to recover the accuracy drop of masked weights: a *quantization range correction*, *the bias compensation*, and *layer-wise optimized masking*.

4.4.1 Quantization Range Correction

As shown in Figure 4.3, assigning forbidden areas into the weights' distribution results into a *band* pattern. The values that can be represented are less, and also the range conversion from floating-point to integers suffers from this operation.
A possible improvement in this direction can be modifying the mapping range in the quantization step to cut off the last forbidden set of weight and increase the

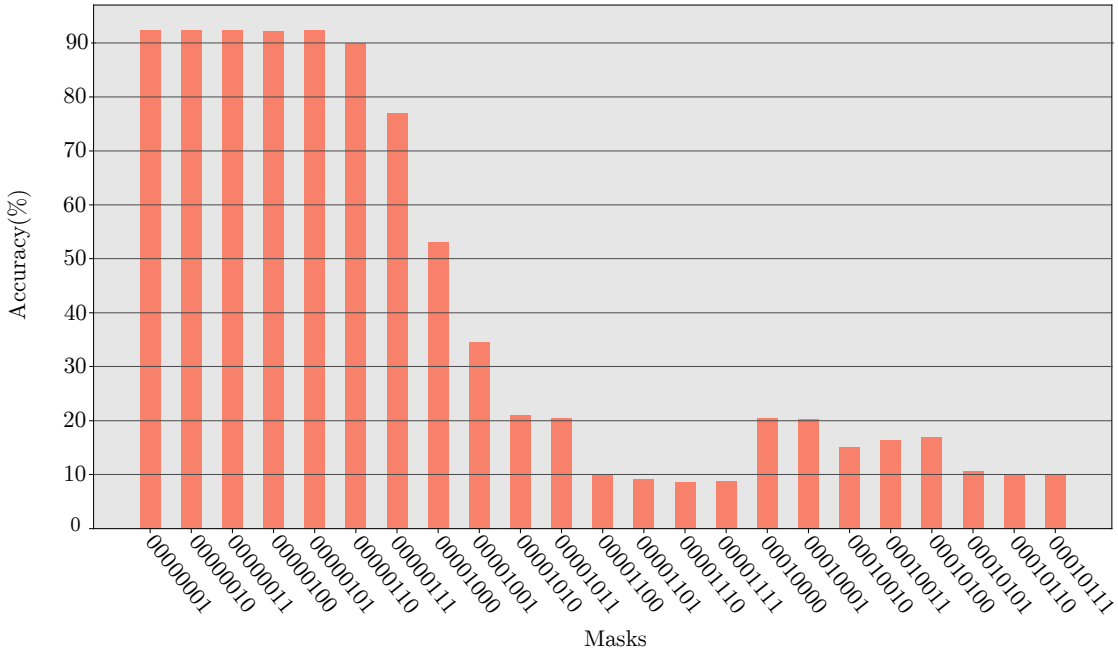

Figure 4.7: Accuracy of masked VGG11 with ARC policy. '1' in the mask string indicates that the bit in that position was permanently fixed to zero.

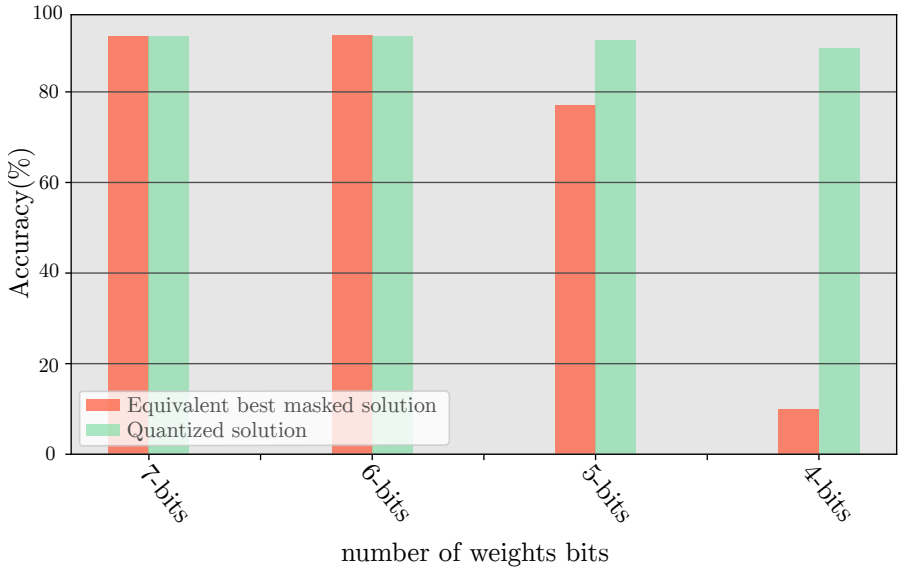

Figure 4.8: Masks compared with different levels of uniform quantization. While aggressive quantization works better decreasing the resolution, masks have a worse impact on the accuracy.

resolution of the output tensor. Mathematically speaking, if the initial quantization formula was:

$$w_q = round\left(\left(w_f - min(w_f)\right)\frac{2^n - 1}{max(w_f) - min(w_f)}\right) \tag{4.5}$$

where $2^n - 1$ was the highest representable value, now it is substituted with the **highest allowed number by the mask**.
For example, in a 8-bit quantized layer where the fifth and the third bit are forced to zero, the value in the formula, from 255 is reduced to 235. Unfortunately, this approach doesn't always improve the model. There are cases in which correcting the quantization range reduces the accuracy or even bias the output toward a specif class prediction. Configurations must be evaluated one by one, and only after checking the accuracy, best solution can be selected.

4.4.2 Bias Compensation

The set of parameter learned during training were those that reached the expected minimum of the loss function. Weights substitution moved the network from the minimum, creating a difference from the reference floating-point model. If kernel's adjacent values are considered, their contribution to the output of each convolution imply an unacceptable difference between the original resulting feature map and the masked one.
To avoid this issue, it was proposed to use a mixed policy in the weight substitution for achieving a zero average error distance. However, it would require extra computation, an ad-hoc optimization algorithm for choosing the right substitution policy at weight level, and an optimal result is still not guaranteed.
The solution adopted, instead, was to use a batch of input data for retrieving layers' activation from both the reference and the masked network. Then the error distance is computed on average for each channel's feature map, and **this value is added to the neuron's bias**.

$$b_k^c = b_k^r + e_k \tag{4.6}$$

b_k^c is the compensated bias applied to the modified model for a specific channel k, b_k^r is the original value and e_k is the average error distance between *all the output computed with the same bias value.* It can be obtained with a specific number of inputs from the test set. From empirical proof network's accuracy doesn't increase over a threashold, which depends on how aggressively the weights have been masked. The optimal number for obtaining the distances was found to be around 500 images,

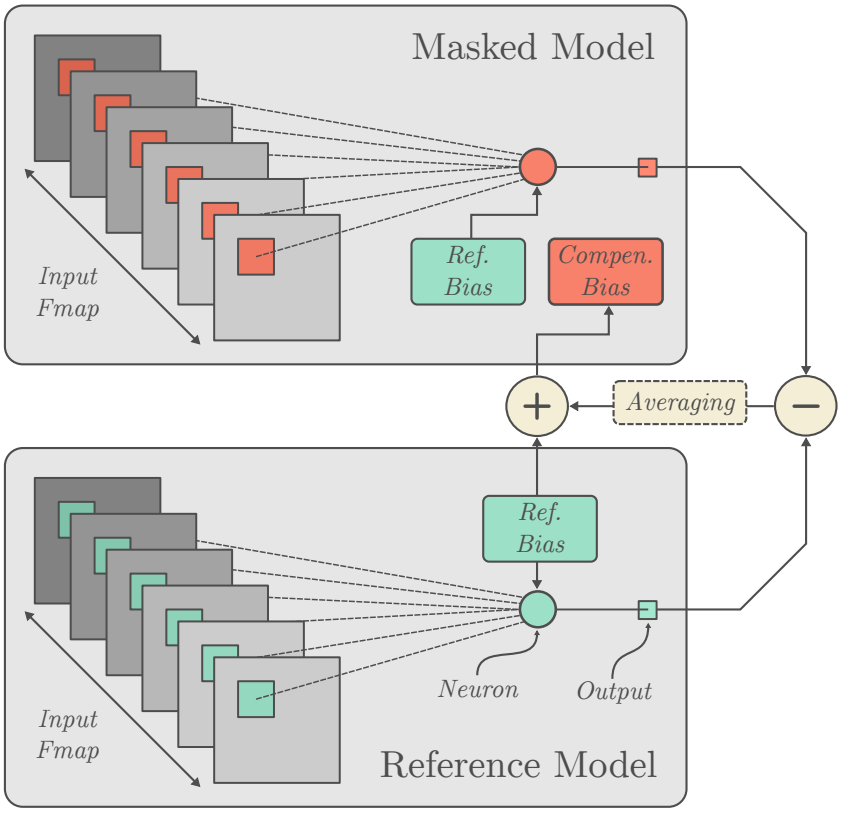

Figure 4.9: *Bias compansation.* In the first step, activations are both computed with the reference bias value. From their difference, the *correction value* is computed and used for updating the bias in the masked model.

which was the value used for all the experiments.

$$e_k = \sum_{i=1}^{N}\sum_{j=1}^{M}(a_{ij}^{ref} - a_{ij}^{mask})\frac{1}{NM} \tag{4.7}$$

Equation 4.7 shows how the correcting error is computed. M is the number of activations in the output feature map, while N indicates how many input samples have been used in the forward pass.
When, for example, biases in the first layer are compensated, input featuremaps of second layer will change. To make this technique effective, layers must be processed one after the other, recomputing the activations at every step, such that every "correction" is aware of all the others done on previous layers.

Acting on the bias can provide significant improvements in the network classification capabilities. In some cases, it increases the accuracy of around 70 percentage points, raising a baseline value of 19% to 88%.

Algorithm 3 Bias compensation algorithm.

Input: reference_model, masked_model, test_batch
Output: compensated_model

```
layer=0
▷ All layer must be compensated
while layer<getNumLayers(reference_model) do
    ref_act = getActivations(reference_model[layer],test_batch)
    mask_act = getActivations(masked_model[layer],test_batch)
    channel=0
▷ Cycling through each output channel bias
    while channel < getNumChannels(masked_model[layer]) do
        ref_act_chs = ref_act[:,channel,:,:]
        mask_act_chs = mask_act[:,channel,:,:]
        e = elementSum(ref_act_ch - mask_act_ch)
        e = e/ref_act_ch.size(0)                          ▷ batch size
        e = e/ref_act_ch.size(2)                          ▷ fmap rows
        e = e/ref_act_ch.size(3)                          ▷ fmap columns
        masked_model[layer].bias = masked_model[layer].bias + e
        channel +=1
    end while
    layer+=1
end while
return masked_model
```

Furthermore, this trick avoids any retraining of the network, which can be time expansive, if several solutions have to be analyzed, or impossible to perform, if the training set is not available. Figure 4.9 shows a graphical representation of the compensation process at neuron's level, Algorithm 3 provides, instead, all the computations required for tuning the whole network biases.

4.4.3 Layer-wise optimized masking

Every mask modifies the weights' distribution differently. Even with correction techniques, layers output may be biased towards a positive or negative direction. Force to zero the same bits uniformly in the whole network is not the best approach. An optimization algorithm that is aware of masks savings and output accuracy is described in Chapter 6.

4.5 Software accuracy results

Figure 4.10 shows some results of *uniform applied* masks to the VGG11 network. Correction techniques are also highlighted to show the improvements that have been achieved in classification accuracy.

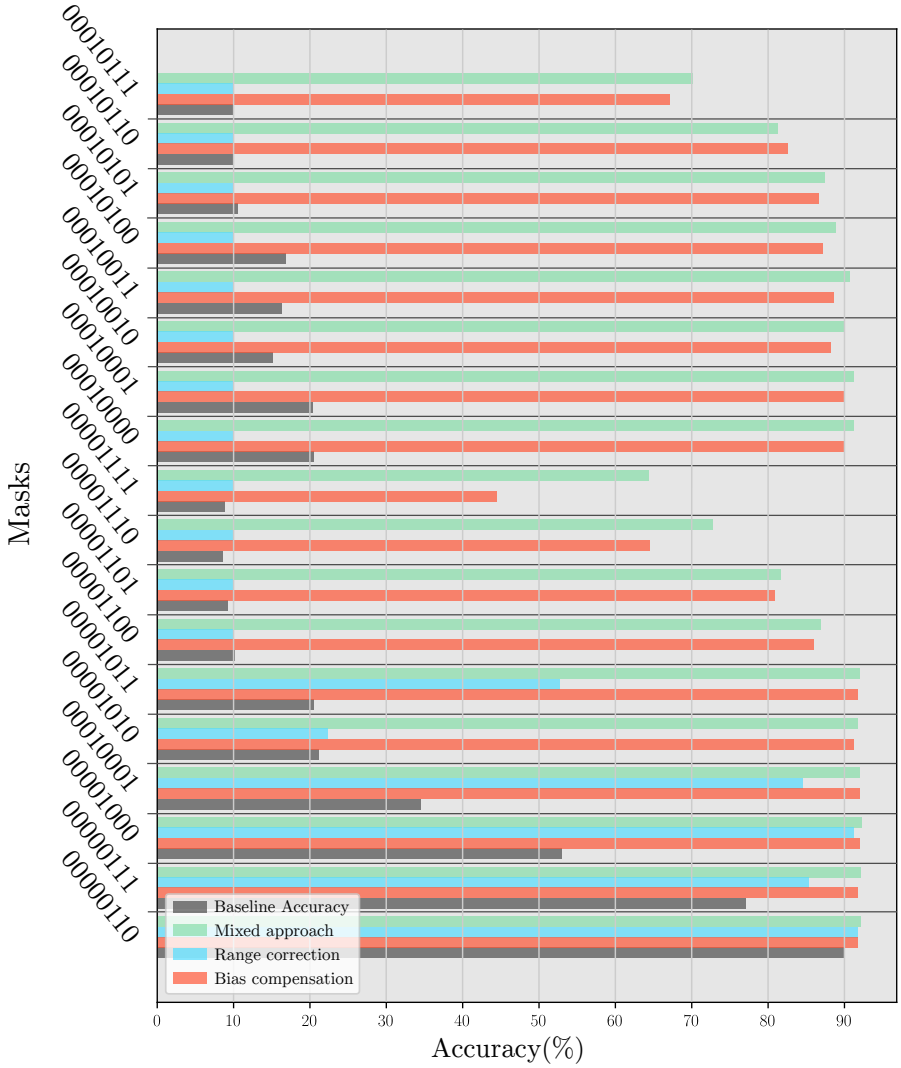

Figure 4.10: Accuracy results on masked VGG11. From the analysis it is clear that correcting the range is not usefull for aggressive masking. For the *mixed approach* the fully connected and the first convolutional layer have been excluded from the masking process.

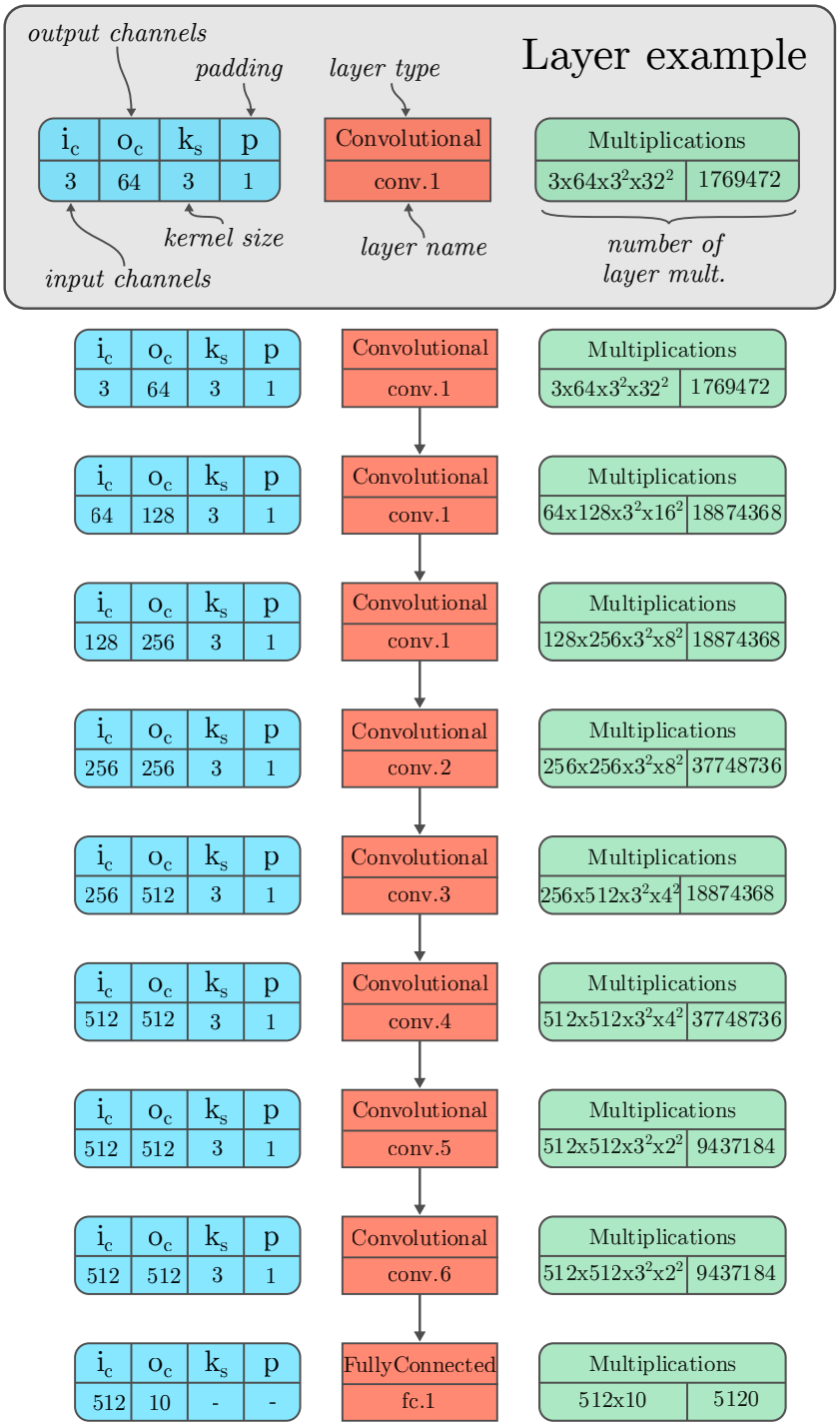

Figure 4.11: VGG11 network description.

Chapter 5

Timing analysis of approximate solutions for the MAC hardware

5.1 MAC Hardware

After improving the accuracy of masked networks, it was fundamental to understand the benefits from the hardware side.
As already said in Chapter 2, state-of-the-art architectures for accelerating convolutional neural networks exploit the systolic array with several layer-mapping approaches. Every PE, independently from the design of memory hierarchy, performs a MAC operation. If the multiplication and the addition are not pipelined, the critical path is the one shown in Figure 4.1.
The choice for this case study was a MAC with unmerged operations: for first data propagate through a *carry-save tree multiplier* (Figure 5.1), and then the result is passed to the adder, for which the implementation is not specified in the design. Baseline architecture receives 8-bits weight and activation, then accumulates the product into a 32-bits partial sum.

With tree-like multipliers, it is straightforward to design specific hardware for a given mask. If the n-th bit of the parameters of a layer is forced to 0, that layer can be mapped into a tile where the logic for handling the n-th partial product is missing in all the multipliers. This without causing an unexpected drop in the network accuracy. An example is provided in Figure 5.2.
Of course, masked models can be mapped into accurate architectures. The same happens between related masks with a different level of aggressivity, allowing more flexibility in case specific designs, made for the proposed approach, are already

available.

5.2 Timing analysis

The goal is to perform a timing analysis for each hardware solution allowed by masks.
Synthesis tools like Synopsys Design Compiler or Cadence Genus can automatically provide optimized solutions if specific inputs are fixed 1 or 0 through a gate-level design by contraction.
For a first analysis, the tool for automated synthesis was used only for choosing the adder implementation, while multiplier architecture was *"hand-designed"* for every mask solution. The motivation behind this choice was the necessity to control the hardware description given as input of the tool.
Since the solution space counted 218 combinations, a Python script was used for producing an ad-hoc CSA tree multiplier for each one of them. Verilog code was then completely verified and sent to synthesis for the timing analysis.
The library used was the *"uk65lscllmvbbr_120c25_tc"* with the Synopsys Design Compiler. The only constraint set was the clock at *0 ns*. The flow is pictured in Figure 5.3, while a qualitative plot of the obtained hardware delays is shown in Figure 5.4.

5.3 Network weighted average delay (WAD)

Thanks to the timing analysis, it was possible to characterize the effect of masks, in terms of data delay, whit respect to the chosen MAC implementation.
But how can these time values be related to the design of the whole neural network? And is it possible to define a function that evaluates the overall gain obtained thanks to masking?
Every layer can be associated with a different configuration, which provides an individual timing improvement. It's also essential to "*weight*" this benefit with the computational load associated with the layer itself. The number of multiplications required for elaborating a complete output feature map is in this case the perfect way to evaluate the impact of every layer configuration on the network performance.
It is possible to define a weighted average delay (WAD) of a model as

$$WAD = \sum_{i=0}^{N} d_{Mk}^{i} \frac{n_{mult}^{i}}{n_{tot}} \tag{5.1}$$

where d^i_{Mk} is the characteristic delay of the mask k associated with the *i-th layer*, n^i_{mult} is the total amount of multiplications performed by this layer, while n_{tot} is the total number of products that the network elaborates for the classification of an image.
The n^i_{mult} value can be obtained for convolutional layers with

$$n^i_{mult} = k_s^2 * i_c * o^2_{c_s} * o_c \tag{5.2}$$

k_s is the kernel size and i_c is the number of input channels. $k_s^2 * i_c$ will be the amount of products computed by a single neuron. o_{c_s} is the size of an output feature map. $k_s^2 * i_c * o^2_{c_s}$ is the number of multiplications needed for producing a complete output feature map; this value has to be counted o_c times, once for each output channel.
The size of output feature maps is given by

$$o_{c_s} = \frac{i_{c_s} - k_s}{s} + 1 \tag{5.3}$$

where s is the stride of the slidings of the kernel.
For fully connected layers, the formula becomes

$$n^i_{mult} = i^2_{c_s} * o_c * ic \tag{5.4}$$

WAD function must be minimized to achieve better performance, but this process can't be done without taking into account network accuracy.
The problem is now a multiobjective optimization, that will be discussed in the next chapter.

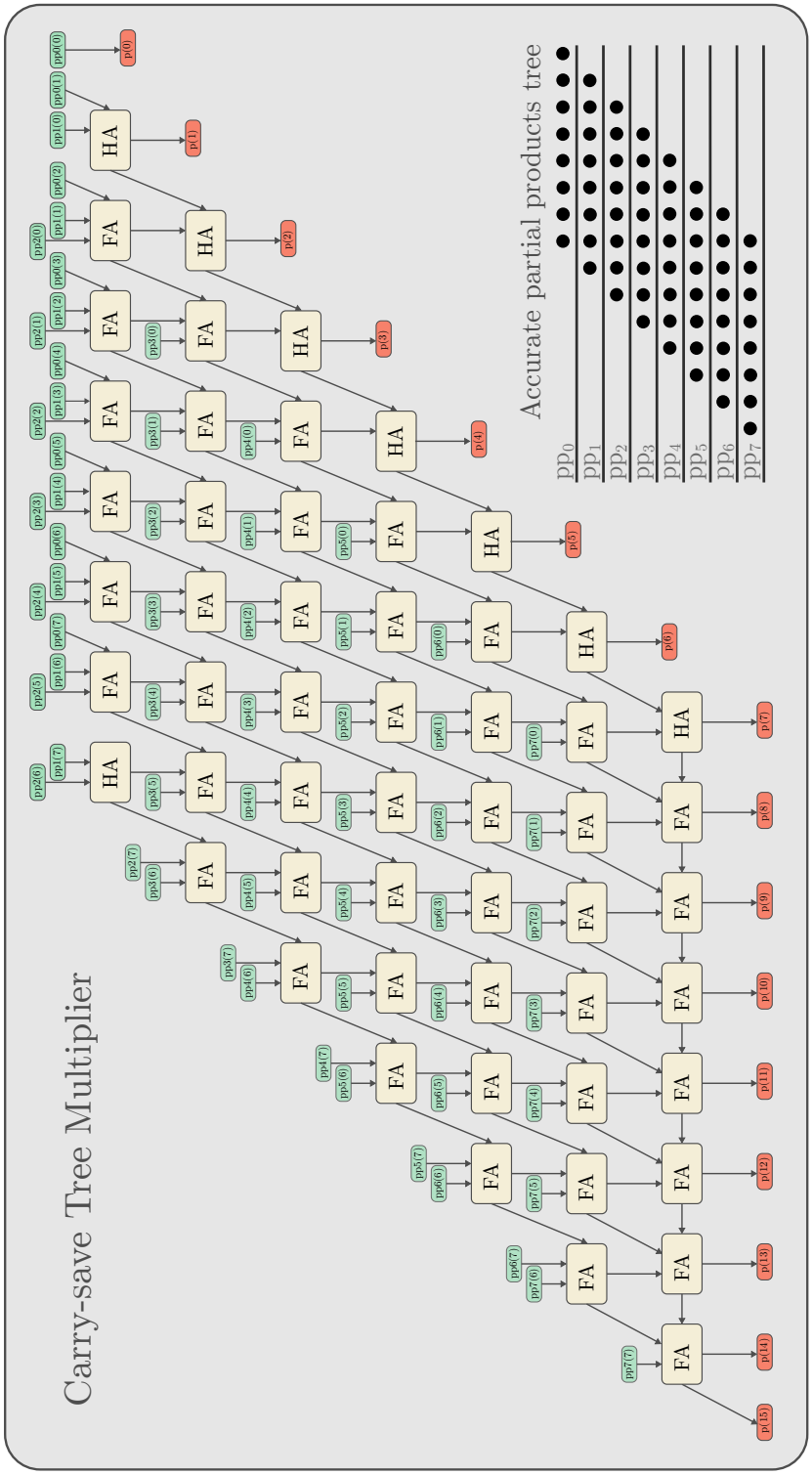

Figure 5.1: CSA tree multiplier architecture.

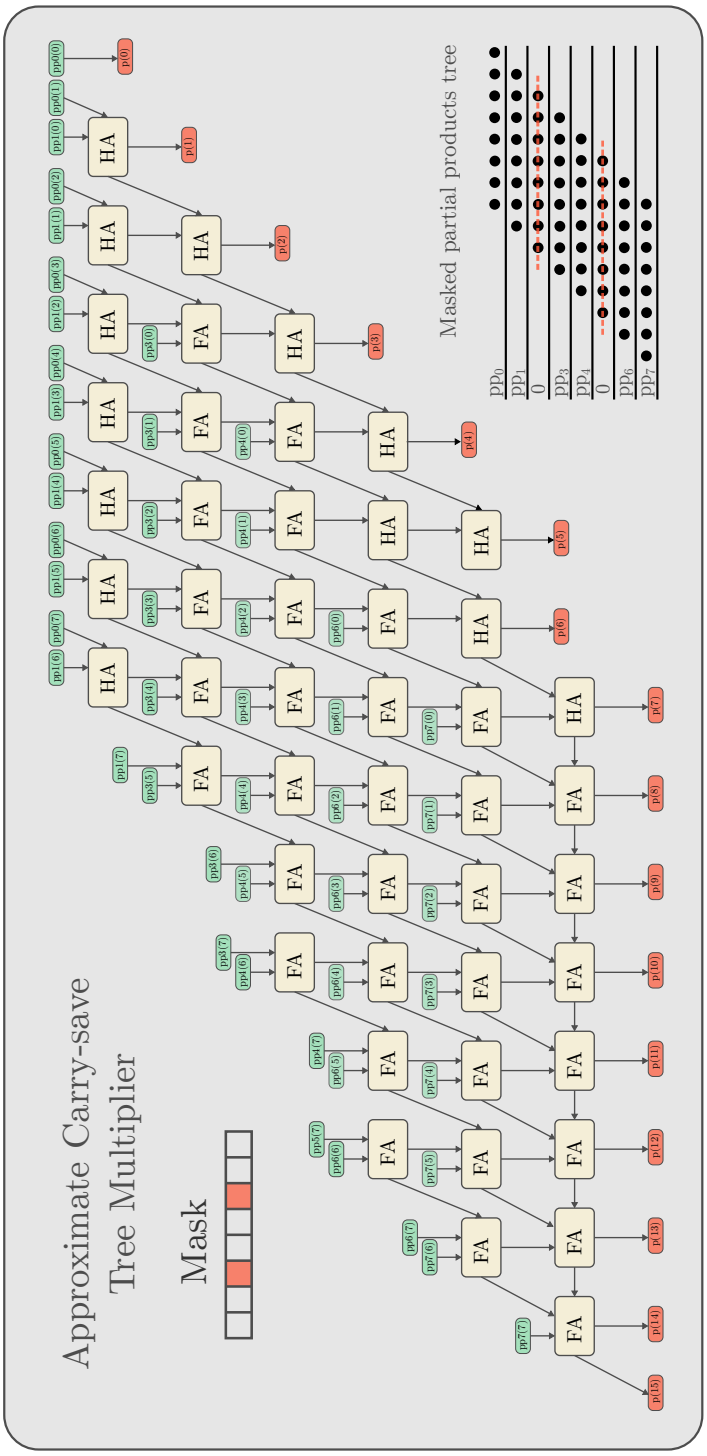

Figure 5.2: Approximate CSA tree multiplier architecture with third and sixth bits masked.

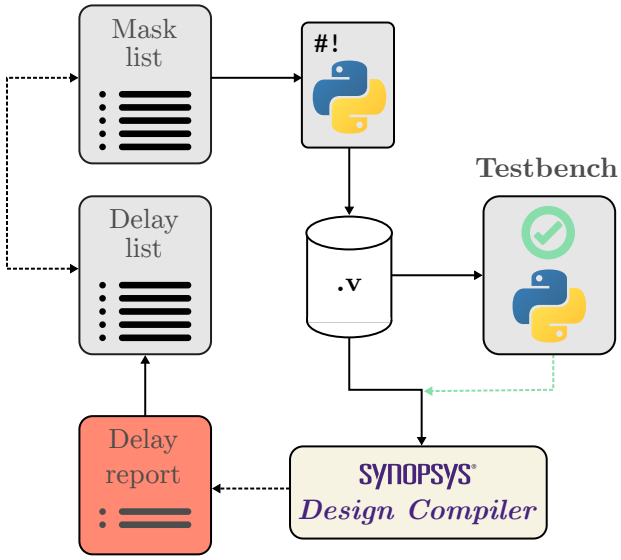

Figure 5.3: Scheme of the mask characterization process. For every configuration (218 total) a verilog file is produced by a Python script. Then the circuit is simulated through Modelsim, results are written to file and then compared to a simple model through another Python script. If outputs are correct, the design is sent to the server with tools for synthesis. Timing report file is then obtained and downloaded. Arrival time is automatically extracted and stored in a new list, which entries correspond to the related masks.

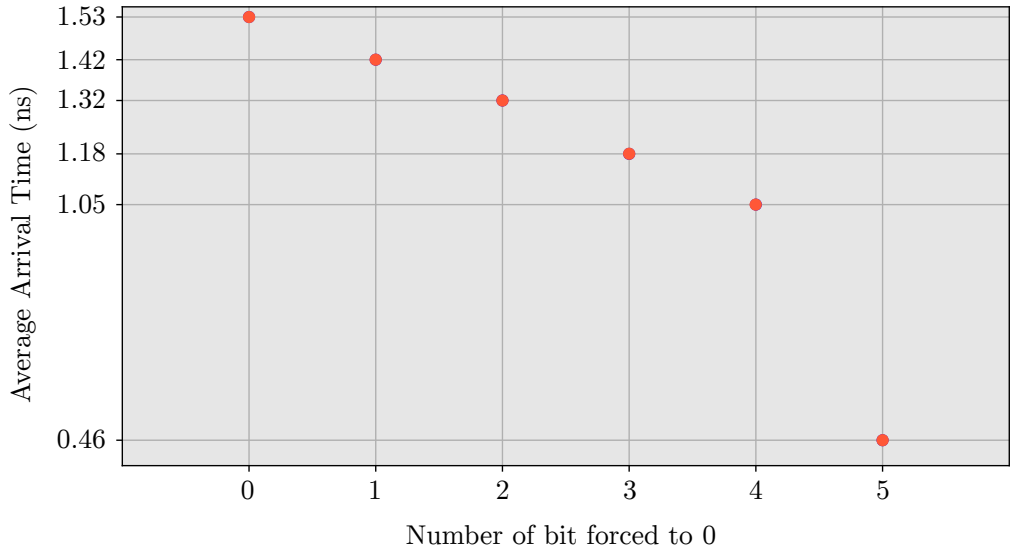

Figure 5.4: Results of *average arrival time* for different level of masking aggressiveness.

Chapter 6

Hardware-Software optimization through the NSGA-II genetic algorithm

6.1 The Multi-objective optimization problem

Every neural network configuration, according to the proposed approach, can be defined with a set of different masks, one for each layer. Accuracy evaluation and WAD function, then, characterize the quality of every solution.
Masks propose a trade-off between accuracy and performance. A decision-maker has to choose which configuration fits most the target application. However, manually exploring the solution space is not trivial.
Let's consider, for example, the VGG11 network, which has eight convolutional layers and a classification one, fully connected. A possible number of masks to choose as candidates for each layer is 32, which is the amount of possible combinations obtained if only the five least significant bits can be masked. If the possibility of correcting the quantization range is taken into account they become 64.
Solution space will be populated by

$$s = l^m = 9^{64} \tag{6.1}$$

which, in this case, offer more than 10^{61} elements. Considering 30 seconds for evaluating each configuration, it would take around 10^{54} years for exploring the whole space. Which, of course, is not feasible and also useless.
Taking a step back, the described problem is a multi-objective optimization where multiple functions, 2 in this case, have to be minimized: top-1 error and WAD.

A variable x_l can be assigned to each layer to express that a mask has to be linked to it. The set of variable is indicated with X_L.

$$f_1(X_L) = forward(network, test\_set) \tag{6.2}$$

$$f_2(X_L) = WAD(network) \tag{6.3}$$

Multi-objective optimization, or *Pareto optimization*, is generally a multiple criteria decision that tries to minimize (or maximize) more than one objective function simultaneously. The optimal choice needs to be taken in case of trade-offs between conflicting objectives, as in this case.
It does not exist a single trivial solution but a set of Pareto optimal configurations. A solution is called *non-dominated* if none of the objective functions can be improved without degrading the other ones. All the points on the *Pareto front* are considered equally good and must be evaluated by a decision-maker.
Another restriction has to be applied here: values for the layers' variables must belong to a confined set of integers, whose dimension corresponds to the number of available masks. Layer l will be assigned to the variable x_l, that will be used as an index to access a look-up table where all masks are stored. Entries of such list will be ordered by increasing characteristic arrival time, that was obtained as described in Chapter 5, Section 5.2. Lower addresses will result in a shorter WAD and, probably, in a more consistent top-1 error.
For a complete solution inspection, each mask configuration can be accessed by two contiguous indexes: even numbers will tell the network mapping function that quantization range, for that specific layer, has to be corrected. The opposite is the case when the address is odd. An exemplification on how this works is provided in Figure 6.1
For a better understanding of how the problem was modeled, a simple example with the LeNet-5 for CIFAR-10 is proposed here in Figure 6.2.
For the sake of simplicity, the last five masks of the table were assigned randomly to a layer. Accuracy is evaluated through the normal inference process, while WAD is obtained as shown in Section 5.3.
For fast exploring space solution to find the Pareto front, the *non-dominated sorting genetic algorithm II* (NSGA II) was used.

6.2 NSGA-II for network optimization

A fast solution space exploration is needed to:

- evaluate what the real potential of masks is, and if they provide a robust alternative to other low-power approaches to CNN design

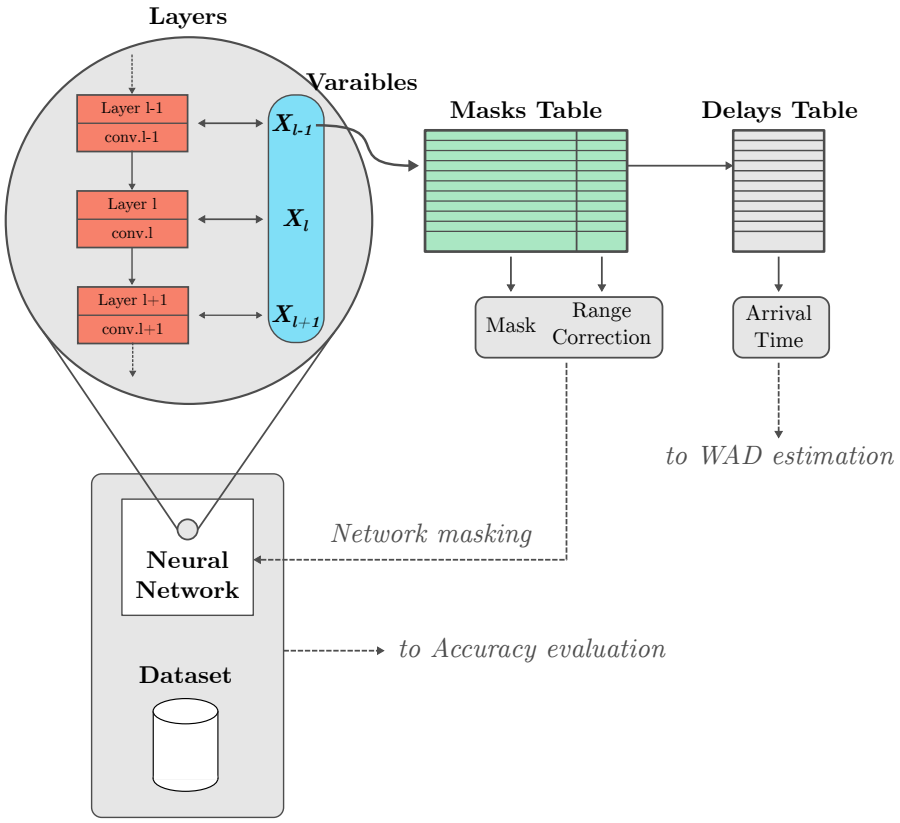

Figure 6.1: The variables associated with layers determine the configuration of the network. Delays will be used for estimating the weighted average dealy, while the model, along with the test set, will be evaluated for obtaining the accuracy.

- understand if valid configurations can be obtained in a reasonable amount of computational time

The option of implementing from scratch an ad-hoc optimization algorithm was avoided. Any a priori hypothesis on how to exclude classes of solutions may result in eliminating potentially suitable candidates.
Heuristic algorithms achieved consistent results in similar circumstances [5], so the choice fell on the NSGA-II genetic algorithm [59]. The implementation can be found on the GitHub repository of pymoo [60] and a simple working scheme is shown in Figure 6.3.
Genetic algorithms are based on Darwin's principle "*survival of the fittest*", where fittest, in this case, are the solutions with the best accuracy and lower delay.
The starting point is a population of different configurations P_t. The fittest individuals from P_t are chosen through a tournament selection; mixing the genes of higher rank solutions, a second set Q_t is obtained; this operation is called *crossover*

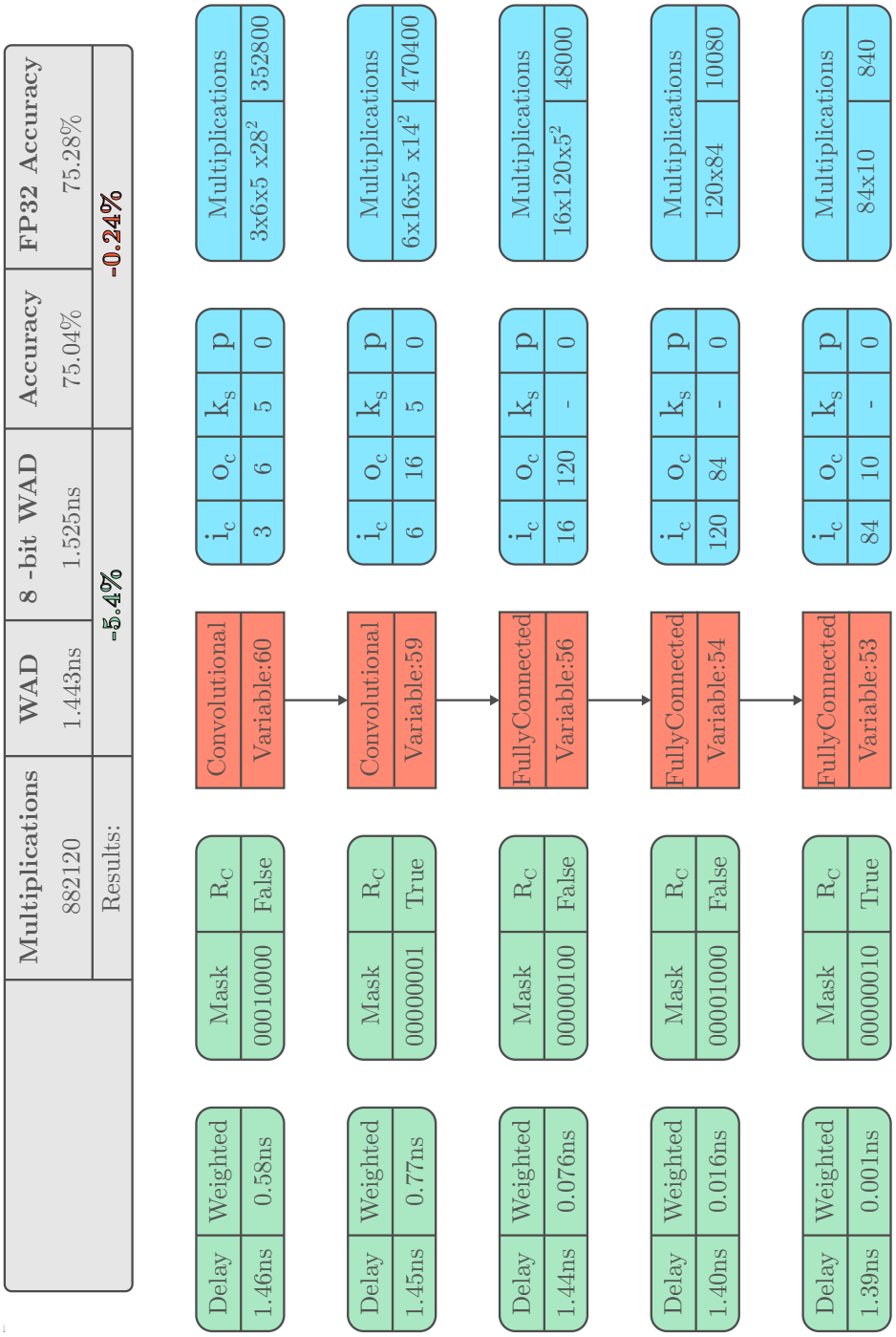

Figure 6.2: Evaluation of a masked example for LeNet-5 on CIFAR-10 dataset.

or *mating*. In the last step before sorting, mutations are applied to the offspring to produce a third set of individuals R_t.
Tournament Selection analyzes the population: individuals that have a high rank are allowed to reproduce in couples. In case rank is equal between two or more solutions, a parameter called crowding distance is evaluated, this tells how "unique" is that solution, or better, how far it is from all the others. Once the parents are selected, crossover takes, according to the mating policy, genes from both and produces child solutions. Lastly mutations may modify some genes of the offspring through a specific procedure, chosen by the programmer.
R_t individuals are evaluated and sorted into three different Pareto fronts with a fast non-dominated approach; this step requires $O(MN^2)$ computational load, where M is the number of objective functions, while N is R_t population size.
Since NSGA-II is an elitist approach to genetic evolution, the first front is directly propagated to the new population along with some solutions of the second and the third, until the number of the P_t individuals is reached. The new set will be called P_{t+1}.
To avoid process to be stacked on local minima or maxima, a crowding distance

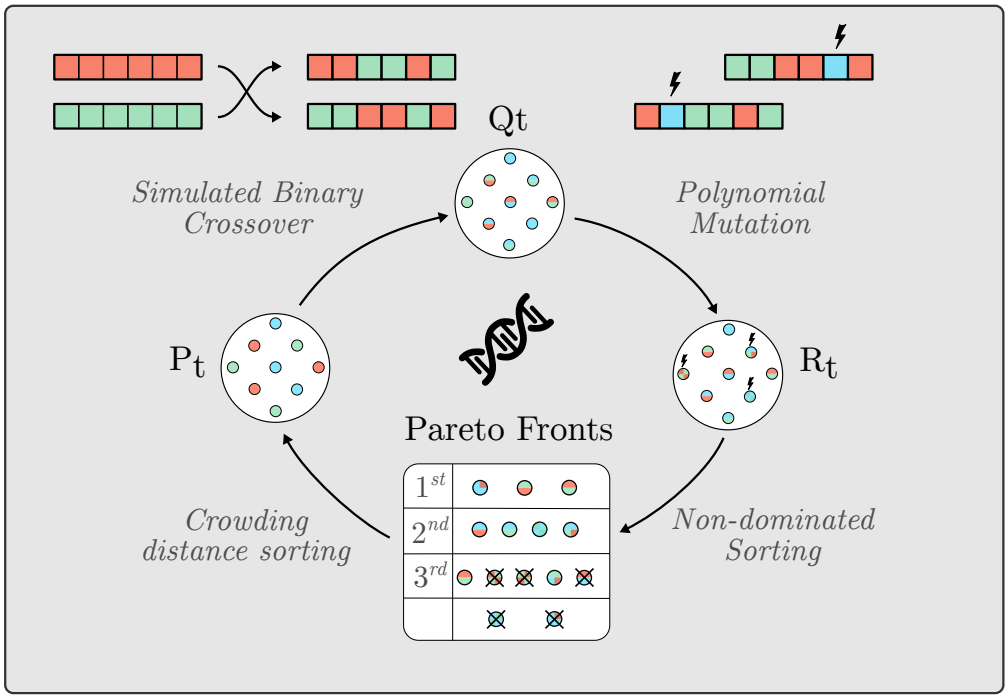

Figure 6.3: Non-dominated sorting genetic algorithm II. Circular scheme of a generation.

sorting is also used to choose the individuals with higher crowding distance from the pool of solutions that doesn't entirely fit in the next surviving population.

Each cycle is known as *generation*. The algorithm can be stopped when the first Pareto front has enough solutions for filling the whole population, or when a definite amount of generations is analyzed.

6.3 Final results

In the execution of NSGA-II for optimizing masks applied to VGG11 network a population of 50 individuals was randomly chosen for the start-up; 100 offsprings were produced each cycle for 200 generations with a total of 19950 models analyzed during the whole algorithm.
For the mating process, the Simulated Binary Crossover policy was used, while Polynomial Mutation was adopted for the child variations.
Results obtained with this parameter configuration is shown in Figure 6.4.

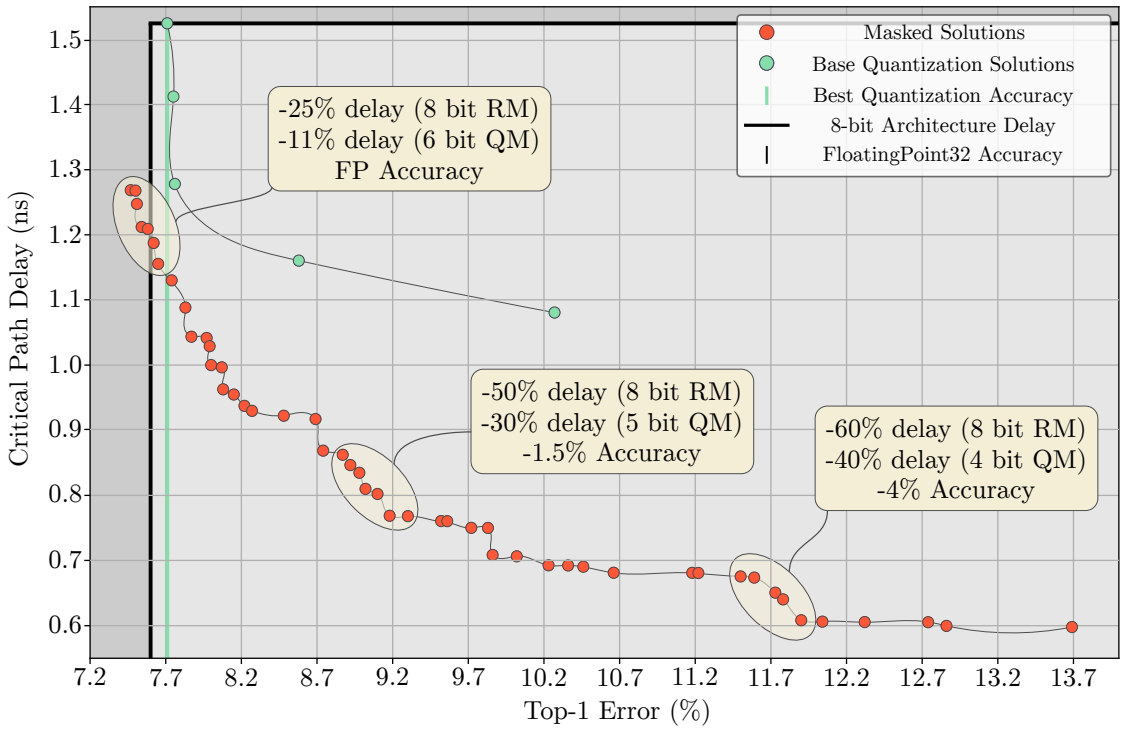

Figure 6.4: Resulting population of layer-wise optimized masked solutions compared with uniformly quantized networks with decreasing number bits used for the weights. RM indicates the Reference quantized Model, while QM the closest Quantized Model.

Conclusion and future work

This thesis provides a versatile yet effective approach to the design of fast PE in deep convolutional neural network hardware architectures.
One of the significant features of this technique is the possibility of mitigating the effects of the approximate hardware with low computational-load tunings, which deliver a broad spectrum of solutions. These sets of configurations are suitable either for safety-critical applications or for several levels of error-resilient contexts. Masks can be applied during the training or, as proposed in this work, after the network has obtained state-of-the-art accurate parameters. Furthermore, other techniques for network distillation, such as pruning and memory access reductions, can be exploited in parallel allowing a further drop in energy and power consumption.
As future work, we would like to link delay surplus with actual energy/power savings as done in [46]. Exploring different PE hardware solutions, along with the use of more network models on different datasets for disparate applications (e.g., object detection, speech recognition, etc.), can also increase the robustness of the proposed technique and demonstrate its validity.
The multi-objective optimization problem of masked networks also created a case-study where different algorithms can be tested and related in order to find the true Pareto-front.
As the last step, we would like to compare our approach with those recognized today as the most effective in the context of deep learning hardware acceleration.

Appendix A

Parameter masking function.

Here is reported the implementation in Python code of the parameter masking function. All the tensor used are of type *torch.tensor*.
_make_mask function returns a number used for forcing to zero the binary digits in all the positions specified in a list passed as an argument.

```
def mask_param(quant_param, bit_to_mask, mask_type=MaskType.SIMPLE_MASK,
                    dynamic=0 , signed=None):
    if bit_to_mask==[]: #case in which nothing has to be done
        return quant_param
    if (dynamic==0 or signed==None) and (mask_type==ARC):
        error_string="'dynamic' and 'signed' parameters must be specified"
        raise ValueError(error_string)

    #operations are defined for integer tensors,
    #but input may be of a different data type
    ty=quant_param.dtype #save original input data type
    quant_param = quant_param.to(torch.int) #cast to int
    if mask_type==MaskType.SIMPLE_MASK: #SIMPLE_MASK case
        mask=_make_mask(bit_to_mask)
        quant_param = quant_param & mask
    elif mask_type==MaskType.ARC:       #ARC case
        mask=~_make_mask(bit_to_mask)
        #generate up_tensor
        #entries of boolTensor will be '1' where number has to be masked
        boolTensor=(quant_param & mask).to(torch.bool).to(torch.int)
        up_tensor=quant_param + boolTensor
        while (boolTensor.sum()): #while there are still velues not allowed
            boolTensor=(up_tensor & mask).to(torch.bool).to(torch.int)
            up_tensor += boolTensor
```

```
25
26          #generate down_tensor (similar to up_tensor)
27          boolTensor=(quant_param & mask).to(torch.bool).to(torch.int)
28          down_tensor=quant_param - boolTensor
29          while (boolTensor.sum()):
30              boolTensor=(down_tensor & mask).to(torch.bool).to(torch.int)
31              down_tensor -= boolTensor
32
33          #exclude overflow numbers
34          max_int = 2**(dynamic-int(signed))-1
35          #entries of overflow tensor will be '1' where number overflows dynamic
36          overflow = (up_tensor>max_int).to(torch.int)
37
38          #computing differences (all positive)
39          diff_up = up_tensor - quant_param
40          diff_down = quant_param - down_tensor
41
42          #boolean tensors for mapping the substitutions
43          #overflows on upper bound tensor must be forced to excluded
44          #NOT is not implemented, ^1 (XOR 1) is used instead
45          up = (diff_up < diff_down).to(torch.int)*(overflow ^ 1) #XOR inverts
46          down = (diff_down < diff_up).to(torch.int) | overflow
47
48          #equal distance mapping, randomly assigning to upper or lower tensor
49          random = torch.ones(quant_param.size()).random_(-10,10).to(torch.int)
50          #converting 0s in 1s, since 0 means here to upper but will be then
51          #interpreted as no need to be masked
52          random = random + (random == 0).to(torch.int)
53          equal = (diff_up == diff_down).to(torch.int) * random
54          up_e = (equal > 0).to(torch.int)*(overflow ^ 1) #0 is not included
55          down_e = (equal < 0).to(torch.int) | overflow
56
57          #executing the actual substitution with boolean tensors.
58          quant_param = up_e*up_tensor + quant_param*(up_e^1)
59          quant_param = down_e*down_tensor + quant_param*(down_e^1)
60          quant_param = up*up_tensor + quant_param*(up^1)
61          quant_param = down*down_tensor + quant_param*(down^1)
62
63      quant_param = quant_param.to(ty)
64      return quant_param
```

Bibliography

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (2017) (cit. on pp. 1, 21, 22).

[2] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. «Analysis and characterization of inherent application resilience for approximate computing». In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013 (cit. on p. 2).

[3] Muhammad Hanif, Alberto Marchisio, Tabasher Arif, Rehan Hafiz, Semeen Rehman, and Muhammad Shafique. «X-DNNs: Systematic Cross-Layer Approximations for Energy-Efficient Deep Neural Networks». In: *Journal of Low Power Electronics* (2018) (cit. on pp. 2, 41).

[4] Vojtech Mrazek, Muhammad Hanif, Zdenek Vasicek, Lukas Sekanina, and Muhammad Shafique. *autoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components*. 2019 (cit. on p. 2).

[5] Vojtech Mrazek, Zdenek Vasicek, Lukas Sekanina, Muhammad Hanif, and Muhammad Shafique. «ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining». In: 2019 (cit. on pp. 2, 31, 38, 39, 41, 65).

[6] Xin He, Liu Ke, Wenyan Lu, Guihai Yan, and Xuan Zhang. «AxTrain: Hardware-Oriented Neural Network Training for Approximate Inference». In: (2018) (cit. on pp. 2, 31, 35, 36, 41).

[7] Frederick Tung and Greg Mori. «CLIP-Q: Deep Network Compression Learning by In-parallel Pruning-Quantization». In: June 2018, pp. 7873–7882 (cit. on pp. 2, 31, 33, 34).

[8] Z. Du, A. Lingamneni, Y. Chen, K. V. Palem, O. Temam, and C. Wu. «Leveraging the Error Resilience of Neural Networks for Designing Highly Energy Efficient Accelerators». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2015) (cit. on p. 3).

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on pp. 5, 6).

[10] S Mor-Yosef, Samueloff Arnon, B Modan, D Navot, and Joseph Schenker. «Ranking the risk factors for cesarean: Logistic regression analysis of a nationwide study». In: *Obstetrics and gynecology* 75 (1990) (cit. on p. 6).

[11] Bruno A. Olshausen and David J. Field. «How Close Are We to Understanding V1?» In: *Neural Comput.* (2005) (cit. on p. 8).

[12] Warren S. McCulloch and Walter Pitts. «A Logical Calculus of the Ideas Immanent in Nervous Activity». In: *Neurocomputing: Foundations of Research*. 1988 (cit. on p. 9).

[13] F. Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain». In: *Psychological Review* (1958) (cit. on p. 9).

[14] Bernard Widrow and Marcian E. Hoff. «Adaptive Switching Circuits». In: *1960 IRE WESCON Convention Record, Part 4*. 1960 (cit. on pp. 9, 10).

[15] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. «Distributed Representations». In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. 1986 (cit. on p. 9).

[16] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. «Learning Representations by Back-Propagating Errors». In: *Neurocomputing: Foundations of Research*. 1988 (cit. on p. 9).

[17] S. Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München*. 1991 (cit. on p. 9).

[18] *Wikipedia List of animals by number of neurons*. `https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons`. Accessed: 2020-09-01 (cit. on p. 10).

[19] Kunihiko Fukushima. «Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position». In: 36 (1980) (cit. on p. 10).

[20] Kumar Chellapilla, Sidd Puri, and Patrice Simard. «High Performance Convolutional Neural Networks for Document Processing». In: 2006 (cit. on p. 10).

[21] Ruslan Salakhutdinov and Geoffrey Hinton. «Deep Boltzmann Machines». In: *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics* (cit. on p. 10).

[22] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. «What is the best multi-stage architecture for object recognition?» In: *2009 IEEE 12th International Conference on Computer Vision*. 2009 (cit. on p. 10).

[23] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jurgen Schmidhuber. «Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition». In: (2010) (cit. on p. 10).

[24] Quoc V. Le et al. «Building high-level features using large scale unsupervised learning». In: (2011) (cit. on p. 10).

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: (Jan. 2012) (cit. on pp. 10, 11, 17).

[26] A. Coates, B. Huval, T. Wang, D.J. Wu, A.Y. Ng, and Bryan Catanzaro. «Deep learning with COTS HPC systems». In: *30th International Conference on Machine Learning, ICML 2013* (2013) (cit. on p. 10).

[27] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going Deeper with Convolutions». In: (2014) (cit. on p. 10).

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. «ImageNet: A Large-Scale Hierarchical Image Database». In: 2009 (cit. on pp. 11, 12).

[29] Natalie Stephenson, Emily Shane, Jessica Chase, Jason Rowland, David Ries, Nicola Justice, Jie Zhang, Leong Chan, and Renzhi Cao. «Survey of Machine Learning Techniques in Drug Discovery». In: *Current drug metabolism* (2018) (cit. on p. 11).

[30] Yann LeCun and Corinna Cortes. «MNIST handwritten digit database». In: (2010) (cit. on p. 11).

[31] Han Xiao, Kashif Rasul, and Roland Vollgraf. «Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms». In: (2017) (cit. on p. 12).

[32] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. «CIFAR-10 (Canadian Institute for Advanced Research)». In: () (cit. on p. 12).

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* (1998) (cit. on pp. 13, 44).

[34] Karen Simonyan and Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». In: () (cit. on p. 17).

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: (2015) (cit. on p. 17).

[36] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «MobileNets: Efficient Convolutional Neural Networks for Mobile». In: (2017) (cit. on p. 17).

[37] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. «ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices». In: (2017) (cit. on p. 17).

[38] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. «A Survey of FPGA Based Neural Network Accelerator». In: (2017) (cit. on pp. 17, 18).

[39] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. «Caffe: Convolutional Architecture for Fast Feature Embedding». In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. 2014 (cit. on p. 17).

[40] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: `https://www.tensorflow.org/` (cit. on p. 18).

[41] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: 2019 (cit. on pp. 18, 47).

[42] *Facebook Big Basin*. `https://engineering.fb.com/data-center-engineering/introducing-big-basin-our-next-generation-ai-hardware/`. Accessed: 2020-09-01 (cit. on p. 18).

[43] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. «Ultra Low Power Deep-Learning-powered Autonomous Nano Drones». In: (2018) (cit. on p. 18).

[44] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. *Neural Network Distiller*. June 2018 (cit. on pp. 18, 24, 47).

[45] Alessandro Pappalardo @ Xilinx Research Labs. *Brevitas*. `https://github.com/Xilinx/brevitas`. 2020 (cit. on p. 18).

[46] Jeff Zhang, Kartheek Rangineni, Zahra Ghodsi, and Siddharth Garg. «ThUnderVolt: Enabling Aggressive Voltage Underscaling and Timing Error Resilience for Energy Efficient Deep Neural Network Accelerators». In: (2018) (cit. on pp. 19, 41, 69).

[47] Yann LeCun, John S. Denker, and Sara A. Solla. «Optimal Brain Damage». In: *Advances in Neural Information Processing Systems 2*. Ed. by D. S. Touretzky. Morgan-Kaufmann, 1990, pp. 598–605. URL: `http://papers.nips.cc/paper/250-optimal-brain-damage.pdf` (cit. on p. 23).

[48] Stephen Jose Hanson and Lorien Y. Pratt. «Comparing Biases for Minimal Network Construction with Back-Propagation». In: *Advances in Neural Information Processing Systems 1*. 1989 (cit. on p. 23).

[49] Babak Hassibi, David G. Stork, Gregory Wolff, and Takahiro Watanabe. «Optimal Brain Surgeon: Extensions and Performance Comparisons». In: *Proceedings of the 6th International Conference on Neural Information Processing Systems*. 1993 (cit. on p. 23).

[50] Nikko Ström. *Phoneme Probability Estimation with Dynamic Sparsely Connected Artificial Neural Networks*. 1997 (cit. on p. 23).

[51] Song Han, Jeff Pool, John Tran, and William J. Dally. «Learning both Weights and Connections for Efficient Neural Networks». In: (2015) (cit. on pp. 23, 31).

[52] Yiwen Guo, Anbang Yao, and Yurong Chen. «Dynamic Network Surgery for Efficient DNNs». In: (2016) (cit. on p. 24).

[53] Michael Zhu and Suyog Gupta. «To prune, or not to prune: exploring the efficacy of pruning for model compression». In: (Oct. 2017) (cit. on p. 25).

[54] *High-Performance Hardware for Machine Learning*. `https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf`. Accessed: 2020-24-01 (cit. on p. 25).

[55] *Building a quantization paradigm from first principles*. `https://github.com/google/gemmlowp/blob/master/doc/quantization.md`. Accessed: 2020-24-01 (cit. on p. 25).

[56] Yoshua Bengio, Nicholas Leonard, and Aaron C. Courville. «Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation». In: (2013) (cit. on p. 26).

[57] Song Han, Huizi Mao, and William Dally. «Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding». In: Oct. 2016 (cit. on pp. 31, 32).

[58] M. A. Hanif, F. Khalid, and M. Shafique. «CANN: Curable Approximations for High-Performance Deep Neural Network Accelerators». In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019 (cit. on pp. 31, 37, 38).

[59] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. «A fast and elitist multiobjective genetic algorithm: NSGA-II». In: *IEEE Transactions on Evolutionary Computation* (2002) (cit. on p. 65).

[60] Julian Blank and Kalyanmoy Deb. *pymoo: Multi-objective Optimization in Python*. 2020 (cit. on p. 65).