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

Doctoral Dissertation
Doctoral Program in Computer and Control Enginering (33th cycle)

Hardware-Aware Compression Techniques for Embedded Deep Neural Networks

Matteo Grimaldi

* * * * * *

Supervisors
Prof. Enrico Macii, Supervisor
Prof. Andrea Calimera, Co-supervisor

Doctoral Examination Committee:
Prof. Jose Ayala, Referee, Universidad Complutense de Madrid
Prof. Anupam Chattopadhyay, Referee, Nanyang Technological University
Prof. Andrea Acquaviva, Università di Bologna
Prof. Paolo Garza, Politecnico di Torino
Prof. Eugenio Villar, Universidad de Cantabria

Politecnico di Torino
October 15, 2021

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial-NoDerivative Works 4.0 International: see www.creativecommons.org. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

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

. . . .

Matteo Grimaldi

Turin, October 15, 2021

Summary

The success of the Internet-of-Things (IoT) is not about the amount of data collected but regards the ability to convert raw data to valuable information. In the last years, data management and interpretation have been brought to a higher level of difficulty, as the number of connected IoT devices has grown dramatically. All the raw data collected locally by the IoT sensors need to be processed and evaluated to extract highly informative data. This process, also known as *sensemaking*, consists of complex data-analysis tasks leveraging artificial intelligence algorithms.

These strategies can predict future states just by learning from past experience, which is represented by features processed on IoT-sensor data collections. Thanks to the rapid advancements in deep learning theory, deep Neural Networks, in particular, machines took a closer step towards human intelligence. These solutions are becoming ubiquitous and scalable to different levels, ranging from natural language processing, speech recognition, computer vision, autonomous driving. Usually, the actual solutions on this topic are deployed offline on centralized high-performance data centers, based on cloud platforms, where the distance between the raw sensor data and the computational hardware is critical. This solution suffers from low scalability.

A wide consensus among the research community is assessing that the overcoming of the IoT computational needs will pass through the development of near-sensor data-analytics accelerators, able to process the collected data at the edge, without introducing time latencies and further power consumption due to cloud computing solutions. Near-sensors data analytics is the key to sustain the IoT ecosystem indeed. Objects that can autonomously extract and process information from the physical world will allow the development of real-time classification solutions able to improve the quality of service in several applications, from remote health care and domotics to smart transportation, smart manufacturing, and smart power delivery. However, deep learning algorithms are extremely complex: the processing requires huge power consumption due to considerable storage, memory bandwidth, and high demand for computational resources. The challenge comes here: make deep learning algorithms fit low-power end-nodes of the IoT.

This dissertation aims to investigate hardware-aware compression techniques to facilitate the process of embedding deep learning solutions on resource-constrained

architectures. The goal is to reduce the energy required to run such large deep neural networks on resource-limited devices. More specifically, the main objective of our research is to find the perfect trade-off between the complexity of the deep learning model and the resources available on-chip, without performance degradation during prediction. This is accomplished through the development of novel software-level optimizations able to address these compelling technological demands.

Various compression techniques have been explored in the last decade to bridge this gap: pruning to remove redundant parameters, quantization to reduce their numerical precision, encoding algorithm with sparse-matrix computation to exploit the approximated parameters, are a subset of them. Several strategies tried to merge these and other compression methods to optimize such deep learning algorithms in terms of storage, memory, latency, and power; however, yet prioritizing one aspect to respect others. Moreover, these techniques were often designed without a proper consciousness of the hardware, limiting the compression effectiveness to a theoretical aspect.

With such a premise, this dissertation is organized into three main parts, each of them focusing on a different objective. The first part focuses on statistically oriented compression of neural networks, with particular concerning on new strategies to exploit the natural over-parametrization of these models. It first illustrates a constrained training to enable an effective approximation of the distribution of weights, with a proper encoding scheme it reaches high compression rates. Then, it presents a novel hybrid methodology capable to discriminate between model layers in terms of significance, with the aim to boost the final compression achievements. In the second part, the focus shifts on the hardware awareness of the compression strategies, a crucial feature to meet the real constraints of the deployment. The dissertation first analyzes the optimality of memory-bounded convolutional neural networks, through a smart heuristic able to explore the memory vs. accuracy solution space. Then, it presents a new technique able to empower the processing of n-ary precision networks on general-purpose microcontroller units. At last, it illustrates an adaptive sparse training designed to maximize the compression of storage-bounded networks. In the third part, the scalability of the deep learning models is addressed with innovative solutions to explore the latency vs. accuracy space. In particular, it presents a novel training and compression pipeline for building nested sparse networks: a set of sub-networks enclosed in a unique model able to run-time scale configuration points, during the inference stage.

The techniques proposed in this thesis provide some useful insights into the edge-driven compression of neural networks. For each of these three topics, results show that the aforementioned demand to balance the trade-off between model complexity and available resource can be effectively addressed. We hope that this work may contribute, with other research in this field, to open up more space and help to make artificial intelligence accessible to everyone, improving the quality of life of our next future.

Acknowledgements

Finally, this is the end. Despite all the difficulties, I will always remember these four years of Ph.D., as some of the best in my life. For this, I would like to express my deepest gratitude to the following people.

I would like to thank Prof. Enrico Macii, for giving me the opportunity to join the EDA research group undertaking this Ph.D. journey.

I would like to express my most sincere gratitude to Prof. Andrea Calimera, who has been a true mentor, passionate teacher, inspirational leader, valued colleague, and professional role model throughout my time at Politecnico di Torino. This dissertation would not have been possible without Andrea's constant stream of ideas, encyclopedic knowledge of technical minutia, and patient yet unwavering support.

A big thanks also to all the members of the EDA group, for welcoming me into this big family. Especially to the *Lab-4* mates, for being supportive and often helpful in my research, and even more for just being great friends. I will always treasure the moments spent together, be it in the lab or out for dinner.

To my parents, for their unconditional and endless love. To my entire family, whose I am very proud of belonging to. Without them, I would not be the same person.

Our treasure lies in the beehive of our knowledge. We are perpetually on the way thither, being by nature winged insects and honey gatherers of the mind.

Friedrich Nietzsche, 1887

Contents

List of Tables x

List of Figures xiii

1 Deep Learning at the Edge 1
- 1.1 Need for Model Compression . . . 2
- 1.2 How to Compress a ConvNet . . . 6
- 1.3 Objectives and Contributions . . . 8

2 Background: Overview of ConvNet Compression Techniques 11
- 2.1 Generic Notation . . . 11
  - 2.1.1 Training . . . 12
- 2.2 Pruning . . . 13
  - 2.2.1 Basic Terminology . . . 14
  - 2.2.2 What to Prune . . . 15
  - 2.2.3 When to Prune . . . 18
  - 2.2.4 How to Prune . . . 22
- 2.3 Quantization . . . 27
  - 2.3.1 Quantized Inference . . . 29
  - 2.3.2 Quantize for the Edge . . . 32
- 2.4 Encoding . . . 34
  - 2.4.1 Sparse-Specific . . . 34
  - 2.4.2 Type-Agnostic . . . 36
  - 2.4.3 Inference of Encoded ConvNets . . . 37

3 Statistical-Oriented Training and Compression 41
- 3.1 Motivation . . . 41
- 3.2 A Compression-Driven Training Framework . . . 43
  - 3.2.1 Training ConvNets in a Constrained Space . . . 43
  - 3.2.2 Experimental Results . . . 49
  - 3.2.3 Conclusions . . . 51
- 3.3 Boosting Compression via Layer-Wise Strategy . . . 53

3.3.1 A Greedy Approach for Compressive Training 53
3.3.2 Experimental Results 58
3.3.3 Conclusions 64

4 Hardware-Driven Training and Compression 65
4.1 Motivation 65
4.2 Optimality Assessment of Memory-Bounded ConvNets 70
4.2.1 PaQ: Prune and Quantize 70
4.2.2 Experimental Results 74
4.2.3 Across the Memory-Accuracy Space 78
4.2.4 Conclusions 85
4.3 Arbitrary Bit-width ConvNets on IoT MCUs 86
4.3.1 Memory-Aware Compression 88
4.3.2 Experimental Results 90
4.3.3 Conclusions 95
4.4 EAST: Encoding-Aware Sparse Training 96
4.4.1 Storage-Aware Compression 96
4.4.2 Experimental Results 99
4.4.3 Conclusions 102

5 Latency-Quality Scalable ConvNets 103
5.1 Scalable ConvNets and Their Knobs 103
5.1.1 Static Scalability 103
5.1.2 Dynamic Scalability 104
5.2 Motivation 105
5.3 Run-time Scalable ConvNets via Nested Sparsity 107
5.3.1 Building Nested Sparse ConvNets 107
5.3.2 Experimental Results 111
5.3.3 Conclusions 119

6 Conclusions and Future Directions 121

List of Publications 125

Bibliography 126

List of Tables

1.1 Main features of the Cortex-M IoT MCUs by ARM [1] 6
3.1 Results on CIFAR10. 50
3.2 Results on CIFAR100. 50
3.3 Results for custom RNN trained on IMDb. 51
3.4 Experimental results on CIFAR10. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR. 60
3.5 Experimental results on CIFAR100. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR. 60
3.6 Analysis of the sparsity and bit-width variation across the layers, before and after the compressive greedy training. On the left, the full precision model (FP), on the right the compressed model (CM), both referring to ResNet20 ConvNet trained on CIFAR10 dataset. The input shapes of the layers are (n, c_{in}, k_h, k_w), and (n, c_{in}) respectively for convolutional (Conv) and fully-connected layers (Fc). The height and the width of the kernels are defined as k_h and k_w, the number of input channels as c_{in}, and the batch-size as n. 61
3.7 Analysis of the sparsity and bit-width variation across the layers, before and after the compressive greedy training. On the left, the full precision model (FP), on the right the compressed model (CM), both referring to AlexNet ConvNet trained on CIFAR100 dataset. The input shapes of the layers are (n, c_{in}, k_h, k_w), and (n, c_{in}) respectively for convolutional (Conv) and fully-connected layers (Fc). The height and the width of the kernels are defined as k_h and k_w, the number of input channels as c_{in}, and the batch-size as n. 62
3.8 State-of-the-art comparison of our technique with TTN work [237] with CIFAR10 dataset. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR, and it is reported just for our solution. For each comparison, we report in bold the solution with higher accuracy. 63

3.9 State-of-the-art comparison of our technique with TTN work [237] with Imagenet dataset. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR, and it is reported just for our solution. For each comparison, we report in bold the solution with higher accuracy. 63
4.1 Table of abbreviations. 75
4.2 Overview of the benchmark. Each model is composed by three types of layers: Convolutional (Conv) of shape (c_{out}, k_h, k_w), max-pooling (MaxPool) of shape (k_h, k_w), and fully-connected (FC) of shape (c_{out}). The height and the width of the kernels are defined as k_h and k_w, while the number of output channels is defined as c_{out}. . 76
4.3 List of the development boards adopted to assess the compressed ConvNets. 76
4.4 Optimal vs. hardware-compliant solutions under different memory targets M_t. From left to right there are three main groups of columns: *Pareto*, **PaQ-8**, and **PaQ-16**. The first details the Pareto points P, the second details the hardware-compliant solutions provided by **PaQ** flow using 8-bit, and the third details the hardware-compliant solutions provided by **PaQ** flow using 16-bit. All these solutions are the same shown in the plots of Fig. 4.4. For the hardware-compliant solutions (**PaQ-8**, and **PaQ-16**) we reported also their accuracy distance (lower is better) to the optimal points in the column Δ. Solutions with too high accuracy losses ($\ll$50%) have not been reported. 81
4.5 Training time needed (expressed as the number of epochs) to reach the baseline accuracy for each task. 85
4.6 Summary of the ConvNets used to validate VQ. Each model is composed by three types of layers: Convolutional (Conv) of shape (c_{out}, k_h, k_w), max-pooling (MaxPool) of shape (k_h, k_w), and fully-connected (FC) of shape (c_{out}). The height and the width of the kernels are defined as k_h and k_w, while the number of output channels is defined as c_{out}. 90
4.7 VQ performance obtained on the three benchmarks under analysis. 92
4.8 Top-1 accuracy on CIFAR-10 and weight memory of the dense ResNet-9 after 32-bit floating-point training (FP32), after quantization (Q8), and after LZ4 compression (Q8+LZ4). 99
4.9 Comparison between state-of-the-art weight pruning (WP) and EAST in terms of Top-1 Accuracy (A) and Sparsity (S). The first column M_t indicates the set of FLASH memory constraints used to compress the ConvNets from the original dense version. The value of the final compression rate (CR) needed to meet the target is reported in the second column. 100

5.1 MobileNetV1 - CIFAR-10. Best results for each sparsity level are highlighted in bold. 114
5.2 ResNet9 - CIFAR-100. Best results for each sparsity level are highlighted in bold. 115
5.3 Storage footprint of ResNet9 trained on Cifar100 and MobileNetV1 trained on CIFAR10. 118
5.4 SSD-MobileNetV2. Best results for each sparsity level are highlighted in bold. 119

List of Figures

1.1 Topology of the VGG-16 ConvNet model [180]. 3
1.2 Comparison of model size versus number of operations versus top-1 classification accuracy (bubble size) for several known ConvNets on Imagenet benchmark [110]. Inspired by the work in [13]. 5
1.3 Tools and constraints in deep learning compression. 6
1.4 An Illustrated overview of the dissertation outline. 9
2.1 A generic MLP, before (a) and after pruning (b). 13
2.2 The figures show how pruning influences the distribution of the weights, before (on the left side) and after (on the right) its application. The example refers to a magnitude-based pruning to the last convolutional layer of MobileNet_v2 architecture trained on Imagenet (with 320 input channels and 960 output channels). The 80% of the weights of the layer are pruned. On the right side, the figure the y-axis has been limited for the sake of comprehension, in order to avoid the huge peak representation in 0 (histogram value = 245760). 15
2.3 Most popular levels of pruning granularity, represented in an example layer of shape $\mathbb{R}^{4\times2\times3\times3}$. The *structured* levels are a and b, while the *unstructured* levels are c, d, e, f. The pruned parameters are colored in red. 16
2.4 The polynomial decay sparsity scheduler. The results are shown for 4 different sparsity levels $s_f \in \{60\%, 70\%, 80\%, 90\%\}$, with same scheduler setting: t_0=5e3, s_i=30%, p=3. Each dot marker represents a pruning step, the frequency was fixed at 1000. The light-grey area (on the left) represents the dense pre-training stage, while the dark-gray area (on the right) indicates the mask freeze stage where the sparsity stops to increase. 21
2.5 Before (left) and after (right) weight quantization. The example refers to the last convolutional layer of MobileNet_v2 architecture trained on Imagenet (with 320 input channels and 960 output channels). 27
2.6 Quantized inference (uint8). 30
2.7 Quantization-aware training 31

2.8 Types of sparse encoding. From top to bottom: BitMap 2.4.1, COO 2.4.1, CSR and CSC 2.4.1. 35
3.1 The proposed training flow. 43
3.2 The distribution of the second layer weights of AlexNet trained on CIFAR10: after preliminary training (left), after pruning (center), after σ-ternarization (right). 45
3.3 Visual representation of the solution space. 46
3.4 Weights matrix encoding: (a) pruned matrix, (b) encoding scheme, (c) actual memory mapping. 48
3.5 Pictorial representation of LSTM network structure used on IMDb dataset. 51
3.6 The proposed net compression pipeline. 54
3.7 AlexNet on Imagenet, layers after the sorting algorithm; the sparsity value S is reported for each layer. 56
3.8 Accuracy evolution during the training of VGG-19 architecture on CIFAR10 dataset. The plot shows all the details of the optimization loop. The blue line is the accuracy after each training epoch (Step4), which is the accuracy of the compressed model (one value for each training epoch). The red dotted line depicts the accuracy evolution inside a full training step (Step3 + Step4) to highlight how the model is able to rapidly recover the accuracy loss after each compression step, using just one epoch of retraining. 58
4.1 Cortex-M family: Active power and on-chip RAM size. 66
4.2 Sparsity vs. Accuracy of a compressed 9-layer ResNet under different memory constraints (the labeled numbers). The net is trained on CIFAR-10, then compressed via weight pruning and encoding. The blue dash-dotted line marks the accuracy of the original dense version (140kB). 68
4.3 Framework overview. 71

4.4 Solutions provided by **PaQ** flow showed in the memory-accuracy space for the three tasks under analysis (a) IC, (b) KWS, (c) FER. The green cross marker (P_x) shows the solution with higher accuracy, while the hatched area enclosed by the white dotted curve highlights the plateau region ($\mathcal{T}$), where the solutions have the accuracy loss $\mathcal{L} \leq 0.5$ w.r.t P_x. The yellow line (**Q**) indicates the solutions obtained applying only b-quantization. The red dash-dotted lines define the hardware-compliant solutions generated by **PaQ**, respectively using 8- (**PaQ-8**) and 16-bit **PaQ-16**; these are the implementations deployed on the physical device. The green dotted line connects the Pareto points (P) in the memory-accuracy space, i.e. all the solutions that have superior accuracy w.r.t. all the other points with the same target memory M_t. At last, all the absolute coordinates of the Pareto points and of P_x are collected on the right-side box, for each solution reporting (target memory, bit-width, top-1 accuracy). The right box collects the absolute coordinate of P_x and each Pareto point in the format (target memory, bit-width, top-1 accuracy). . . 80
4.5 Memory footprint vs. Top-1 accuracy for KWS. 83
4.6 Average inference time per sample of PaQ-8 solutions on KWS. . . 84
4.7 A pictorial representation of the comparison between the standard n-ary Quantization (left) and our proposed method, Virtual Quantization (right). The different depths of colors refer to the bit-width. 86
4.8 The Virtual Quantization flow. 87
4.9 Top-1 accuracy loss in the three different applications (lower is better). The baseline accuracies are the same reported in Table 4.7. . . 93
4.10 Weight pruning (a) vs. block pruning (b). Colored weights denotes zero-values. 97
4.11 Comparison between the speeds of EAST and weight-pruning to meet the target. The line plot shows the Memory trend according to the training epochs for weight pruning (blue line) and EAST (red line). The memory target is fixed at $M_t = 32$kB (dashed line). Each black dot marker indicates one increment of the block size. . . 101
5.1 Training step: full weight-set (θ) and the sub-nets (θ^{s_i}) sorted with an increasing order of sparsity from left to right (i.e $s_1 < s_2 < s_3$). 108
5.2 Example of the proposed storage format NestedCSR applied to a 1×2 block sparse matrix that can work in three sparsity levels $S = \{s_1, s_2, s_3\}$. 110
5.3 Latency values normalized for each width to the NestedCSR@s=70%. The latency of the dense model at w=1.00 is not shown as it exceeds the FLASH memory of the adopted device (2MB). 116

5.4 Latency-accuracy scaling for Slimmable ConvNets and Nested Sparse ConvNets. Grey area shows the unfeasible solution space for the adopted MCU, i.e., FLASH footprint $> 2MB$. 117

Chapter 1

Deep Learning at the Edge

The Internet of Things (IoT) is one of the most prominent trends in technology to have emerged in recent years. Its success is not about the amount of data collected, but it regards the ability to automatically convert raw data to valuable information. In the last years, data management and interpretation have been brought to a higher level of difficulty, as the number of connected IoT devices has grown dramatically. All the raw data collected locally by the IoT sensors need to be processed and evaluated, to finally extract highly informative data. This process, also known as *sensemaking*, consists of complex data-analysis tasks leveraging machine learning (ML) algorithms.

These approaches are able to predict future states just by learning from past experience, which is represented by the features processed on IoT-sensor data collections. In particular, one branch of ML algorithms has been exploring deeply for these applications: this is Deep Learning (DL) and its most popular algorithms are Neural Networks (NNs). These techniques are becoming ubiquitous and scalable to different levels, ranging from natural language processing, speech recognition, computer vision, autonomous driving. The sensemaking process is commonly implemented as a cloud service, where the DL solutions are deployed offline on centralized high-performance data centers, based on cloud platforms. However, the distance between the raw sensor data and the computational hardware is critical, making these solutions suffer from low scalability.

A wide consensus among the research community is assessing that the overcoming of the IoT computational needs will pass through the development of near-sensor data-analytics accelerators, able to process the collected data at the edge. Objects able to autonomously extract and process information from the physical world will allow the development of real-time classification solutions able to improve the quality of service in several applications, from remote health-care and domotics to smart transportation, smart manufacturing, and smart power delivery. The edge DL paradigm is the key enabling technology to improve the scalability of the IoT

ecosystem, as it ensures real-time responses, lower energy consumption, and higher data privacy.

The cloud-to-edge shift is not free-lunch, however. Off-the-shelf NNs are incredibly complex: the processing requires huge power consumption due to considerable storage, memory bandwidth, and high demand for computational resources. Edge DL implies processing these complex algorithms in a mW power envelope using tiny processor cores with limited computational resources and low storage capacity. This sets a clear limit to the complexity of Neural Networks that can be hosted. Here the main challenge comes: make DL algorithms fit low-power architectures for the IoT edge.

This dissertation proposes novel compression techniques for embedded deep neural networks to run on the edge of the IoT domain. The goal is to reduce the energy required to run such large deep neural networks on resource-limited devices. The techniques explored in the following chapters are focused on the software level optimization of such networks, with the aim to enable their edge migration improving the energy vs. quality trade-off.

In the rest of this chapter, we first overview the complexity and computational load of NNs and then analyze the most popular techniques to bring these solutions to the edge.

1.1 Need for Model Compression

Recent forecasts on internet traffic provided by Cisco annual report [39] suggest that by 2023 the number of connected devices will be 28.3 billion, which is more than three times the global population. The same number was 18.4 billion just in 2018. Undoubtedly, such speed of a huge amount of data represents a volume that communication and storage infrastructures will not be able to afford. Due to the pervasiveness of smart systems in our everyday life, performing complex machine learning tasks on remote cloud workstations will soon become unsustainable. With such a premise, it is clear that smart digital devices need to be equipped with embedded machine learning capabilities. Unfortunately, this hardware migration represents one of the biggest challenges for the design automation community.

From the hardware perspective, the most popular solutions for the edge inference of NNs can be grouped into three categories: (i) custom accelerators based on application-specific designs (ASICs), (ii) general-purpose embedded CPUs, and (iii) ultra-low-power microcontroller units (MCUs). These are different solutions for edge deep learning, each one may be more suitable for a different use case; in fact, there is not an optimal and generic solution for all the scenarios. Custom ASICs guarantee performance stability and power efficiency, but off-the-shelf NNs are static models not able to fully take advantage of the ASIC reconfigurability. Embedded CPUs, usually involved for mobile devices, offer higher flexibility than

custom hardware but are not very suitable for low-cost IoT applications. At last, MCUs are the most low-power solution for edge deep learning, which usually require just a few mWs power to run. They are very appealing for the IoT scalability, but the migration of NNs on them is challenging both for limited resources and for the limited instruction set, as highlighted in the following chapters.

Automatic classification is one of the most relevant activities in the field of machine learning. In a nutshell, a classification problem arises when an object needs to be assigned to a class based on the values assumed by some of its attributes. As far as *structured data* are concerned, statistical and mathematical models like Support Vector Machines [82], and Classification and Regression Tress [11] are preferred due to their lightweight computational requirements and undoubted effectiveness. However, with the increasing need of analyzing huge amounts of *unstructured data*, e.g., pictures with ambiguities like road signs for autonomous vehicles or noisy audio signals for track isolation, Convolutional Neural Networks (ConvNets) have become the DL *swiss-knife* for accomplishing complex classification tasks.

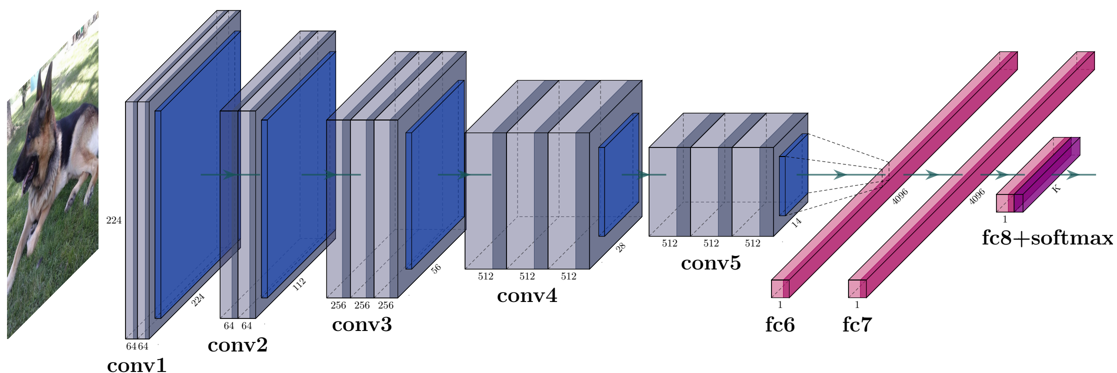


Figure 1.1: Topology of the VGG-16 ConvNet model [180].

A ConvNet is an artificial system that recreates the mechanisms regulating the primary visual cortex of the human brain [105]. Three are the most important instruments: (*i*) *local connections* guarantee a specific neuron's connectivity to belong to a limited region of the analyzed image; (*ii*) *layering* represents the possibility of abstracting more and more complex information like the depth of the network structure grows; and (*iii*) *spatial invariance* allows to detect objects with different shapes, colors, and positions. Figure 1.1 depicts a typical ConvNet structure. Two main logical regions can be identified: a feature extraction region composed of an input layer and several hidden layers (either convolutional, pooling, or dropout layers) that handle n-dimensional input images computations, and a classification region (fully connected, softmax, or dense layers) that is in charge of producing the final answer to the classification problem.

Although each layer has unique functionality, they all perform the same mathematical transformation between an input signal, expressed with a tensor $\vec{x}$, towards an output tensor $\vec{\phi}$. Indeed, each neuron represents a matrix-vector multiplication function as the one reported in Equation 1.1, being θ the matrix of the weights, $\vec{b}$ an adjusting offset parameter (bias), and $f(\cdot)$ an activation function.

$$\vec{\phi} = f(\theta\vec{x} + \vec{b}) \tag{1.1}$$

Both θ and $\vec{b}$ are learnable parameters. During the training phase, they are constantly tuned to adhere to the n-dimensional shape of input functions in order to converge to a feasible and reliable solution to the classification problem. A typical ConvNet is composed of several cascading layers, each of which has thousands, or even millions, of different weights. For instance, the VGG-16 architecture (Figure 1.1) that won the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [110] is composed of 19 layers containing a total amount of 140 million weights. Most computational intensive layers are the convolutional (average of 2 million parameters) and fully connected ones (average of 41 million weights).

The approach for ConvNets design is radically changed during deep learning evolution. Indeed, in the first period of deep learning the ConvNets were designed just from an accuracy-driven perspective: the authors used to increase the number of parameters and relative operations of the models. This *old-family* of ConvNets was proposed without any consideration about the computation efficiency, but only with the aim to increase the prediction capabilities. Then, in the second period, the authors started to take into consideration also the ConvNets portability in environments with limited resources, hence exploring new design approaches able to care about the practical trade-off between accuracy and complexity. The main knobs involved to tune the design process were two: the depth and the width. The former scales the number of ConvNet layers, the latter modules the number of channels (or filters) of each convolutional layer. The depth was firstly introduced by VGG networks [180], while the width was introduced with Mobile Networks [85].

Figure 1.2 reports a visual comparison between most popular ConvNets trained on ImageNet [110]. Looking at the picture, two are the main observations that can be drawn. First, the bubble plot in Figure 1.2 clearly shows that most of the standard ConvNets have more than 10M parameters with more than 1K operations. This assays that moving off-the-shelf ConvNets to the hardware level is a challenging task. Accurate ConvNets on complex tasks require a huge amount of computational firepower, as well as very large memory banks that usually represent serious bottlenecks in terms of speed and energy efficiency. For example, as reported in [13], the VGG-16 model requires about 12W and 800MB of memory on an NVIDIA Jetson TX1 board. The same ConvNet cannot be deployed on tinier hardware devices, like the ones powered by MCUs of the Cortex-M family

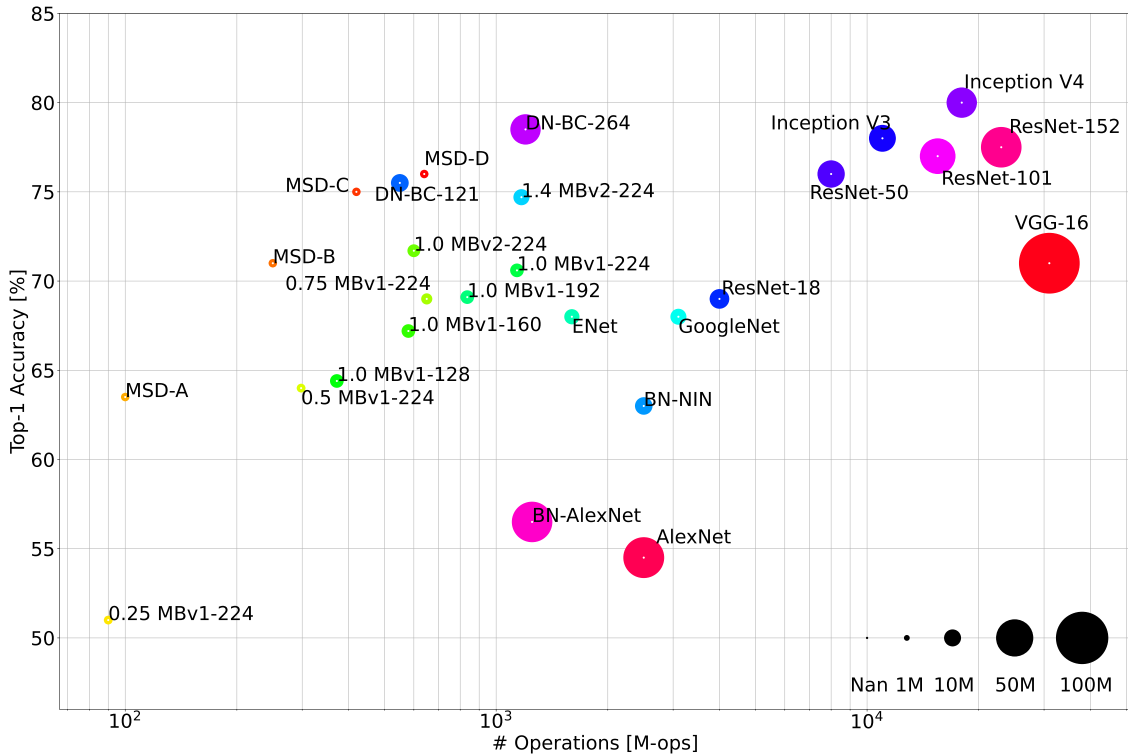


Figure 1.2: Comparison of model size versus number of operations versus top-1 classification accuracy (bubble size) for several known ConvNets on Imagenet benchmark [110]. Inspired by the work in [13].

by ARM[1]. As reported in Table 1.1, they are equipped with few kBs of on-chip (ranging from 4 to 512 kBs, depending on the chip-set), which is insufficient to run most of the architectures reported in Figure 1.2.

Second, a bigger ConvNet does not always mean a better ConvNet. Artificial neural networks in general are systems trained on examples. Their effectiveness is not only affected by the quality of the adopted training set but also by the generalization capabilities that a model is capable to carry out. Indeed, from a quality-of-result perspective, the main concern is how well a ConvNet generalizes patterns outside of the training set. Having more parameters does not always guarantee an improved classification accuracy, as Figure 1.2 reports. Overfitting and local minima represent two critical conditions usually determined by over-parameterized networks, thus preventing fruitful exploitation of ConvNets capabilities. Therefore, a perfect generalization system would be the one with the smallest size that best fits the shape of the data. Needless to say, selecting the optimal ConvNet size is

[1] https://os.mbed.com/platforms/

Cortex-M	Power (μW/MHz)	RAM (KB)	Floating (32b)	Integer (16b,8b)	SIMD Unit (#lane)
M0	5.3	4-32	No	Yes	No
M3	11.0	32-128	No	Yes	No
M4	12.3	128-256	No/Optional	Yes	2
M7	33.0	256-512	No/Optional	Yes	2

Table 1.1: Main features of the Cortex-M IoT MCUs by ARM [1]

not an obvious task. A clear example of this phenomenon is the comparison between VGG16 and MobileNet_v1 architectures. VGG16 belong to the *old-family* of ConvNets, which are strongly overparametrized with a high number of channels per layer, while MobileNet_v2 is a more recent architecture belonging to the *new-family* of models designed to better fit resource-constrained environments, with high depth and small width. Despite VGG16 is 37.7× bigger than MobileNet (88MB vs. 14MB), they both reach 71.3% of accuracy on the same task.

Both the observations are critical aspects of current research on efficient deep learning. Both can be mitigated by optimization and compression of the ConvNets structures. The rationale is that many parameters in the network structure are redundant, thereby inducing noise inside the model, as well as an increased algorithm complexity with higher memory requirements. Very deep structures are usually preferred to have a general and robust model less sensitive to initial conditions. However, too many irrelevant parameters are harmful to prediction performance and resource usage. This motivates the need of effective compression techniques to facilitate the edge deployment of these models.

1.2 How to Compress a ConvNet

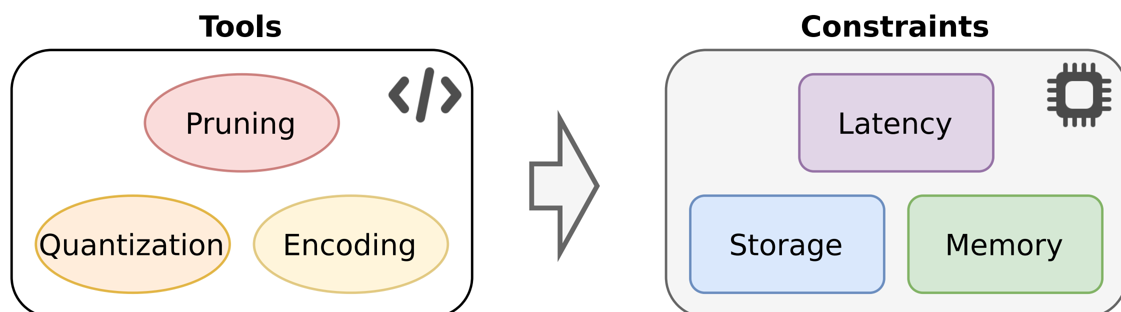


Figure 1.3: Tools and constraints in deep learning compression.

The interaction between algorithms and hardware is a crucial aspect to compress a ConvNet. To define a compression pipeline there are two main choices to fix: what is the constraint to optimize and which tools to use. The former question regards where the ConvNet needs to be optimized (i.e., storage, memory, latency) to be run on a specific hardware device, while the latter question indicates which software tools (i.e., for example, pruning, quantization, encoding) can address these hardware demands. Figure 1.3 shows a schematic view of both the tools and the constraints. Usually, the choice of hardware device where to run the ConvNet automatically implies the target constraints (one or more) of the compression, while the designer has to choose which tools to involve to reach the objective, possibly with no accuracy loss. The tools represented in Figure 1.3, that are pruning [72], quantization [28], and encoding [67], are not the only ones presented in the literature, but just a subset of them we selected as the most used in the works presented in this dissertation. We suggest referring to the state-of-the-art survey in [31] for a more exhaustive overview of the available tools for ConvNet compression. All these three tools are software side optimization of ConvNets, but they usually require some hardware co-design strategies to fully accomplish their benefits. Pruning removes redundant connections from the ConvNet: it downsizes the original topology when applied as structured-level, while it induces sparsity inside the tensors with no reshaping when applied as unstructured-level. Data quantization reduces the data bits used to represent the parameters, which originally are stored in 32-bit full precision. Encoding reorganizes the storing format to exploit eventual common patterns present in model tensors, like sequences of neighboring zero values. In the following Chapter 2 we further provide a more detailed description of these three popular strategies.

Depending on the complexity of the task, and by the strictness of the constraint, one single or a combination of multiple compression tools should be used. For example, if the objective is to reduce the memory storage of a model, we may use just 8-bit quantization to reduce the storage footprint by 4× if the constraint is too stringent; or instead, we may combine sparse pruning, quantization, and encoding to boost the compression in order to meet a more stringent constraint. All these choices strongly depend on the constraints and compatibility of the target device. Interestingly, these approaches with all the possible combinations produce different trade-offs in terms of application scenario, compression ratio, model accuracy, and hardware usability.

In the same as for compression tools, also the constraints showed in Figure 1.3 are not the only features to optimize for edge inference, but for sure latency, storage and memory are the most popular, and also the ones we focus on the works will be presented in the following chapters of this dissertation. For each edge device, the lack of storage space is relevant, indeed even the smallest ConvNet requires tens of MBs of space, while low-power devices usually have limited space. Especially low-power MCUs suffer from this issue, as they usually are equipped just

with less than 2MB of non-volatile memory. Another important feature to consider for efficient edge inference is the capacity of the device to compute complex operations, which depends on the on-chip random-access memory (RAM). Usually, for the IoT segment, RAM available in low-power is limited to a few hundred KBs, as for example the Cortex-M MCUs by ARM reported in Table 1.1. A ConvNet compression focused to reduce the computation memory required to process a certain task would be extremely beneficial to the MCU working. At last, some applications, like keyword spotting or object detection, require fast inference time to process input samples, limiting the inference latency to fixed to a certain range of times. To meet these requirements possible solution is to induce sparsity through pruning inside the ConvNet. Then using custom kernels for matrix multiplication the inference can sensitively speed up, which may be crucial to respect latency constraints. This strategy can be applied differently across processors, ranging from microcontrollers [225] to GPUs [38].

1.3 Objectives and Contributions

Until now there is no a standard and optimal compression pipeline able to optimized a ConvNet model for efficient edge inference from all the points of view. Several strategies have been proposed in the last decade, but they tend to prioritize one aspect respect others. Furthermore, most of these techniques, especially in the first years, were designed without hardware awareness, limiting the compression benefits to a theoretical aspect. In this context, this dissertation has a threefold objective.

- Developing a novel software-level optimization strategy to improve the effectiveness of ConvNet compression. Focusing on how to exploit parameters redundancy with proper encoding strategies is possible to boost the compression with negligible prediction degradation. The objective is to provide a more compact model: easier to deploy and to run at the edge.
- Searching optimal solutions in several constrained solutions spaces. Different compression strategies induce different benefits to the edge device, with a not homogeneous distribution between storage, memory, and latency. One strategy to rule all the constraints is not trivial, but a full-stack design with hardware awareness improves radically the efficiency-quality trade-off.
- Exploring automated routines to scale ConvNets at run-time. The staticity of ConvNets can be overcome using proper knobs, like width, depth, and pattern-based sparsity. These strategies are demonstrated to be very effective and promising for future real-case scenarios in the IoT domain, as they allow to provide multiple solutions still training a unique ConvNet model.

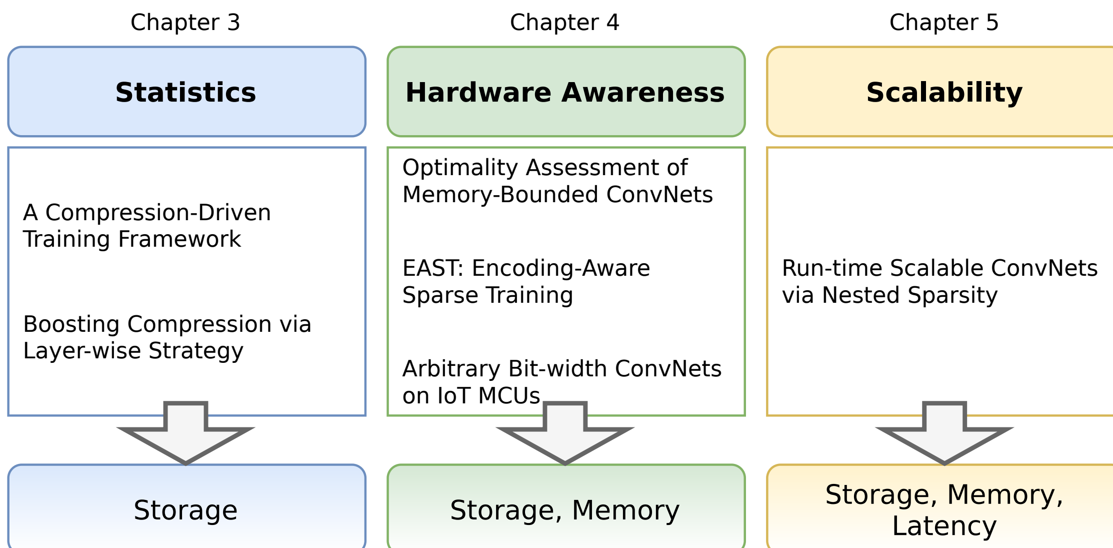


Figure 1.4: An Illustrated overview of the dissertation outline.

For the sake of comprehension, Figure 1.4 illustrates a brief summary of the technical contributions enclosed in this thesis. The three technical chapters (3, 4, 5) are preceded by a brief survey Chapter 2 to overviews the most popular compression tools. For each technical chapter, we presented various strategies to optimize the ConvNet models targeting three main aspects: (i) the storage (i.e., the area dedicated to long-term store the model parameters), the memory (i.e., the area dedicated to short-term allocations used during model processing), and the latency (i.e., the time delay between cause and effect of the network inference).

Chapter 3 targets on *statistically* oriented compression, focusing the attention on weights distribution and how to compress them. We first studied the behavior of deep ConvNets during the learning phase, particularly how the distribution of their parameters changes during training. Then we focused on their natural over-parametrization [23], and how to leverage such redundancy through state-of-the-art compression techniques. We drove the compression pipeline with a homogeneous fusion of pruning [72], quantization [28], and encoding [67] during the training loop. The aim was to enhance the accuracy vs. compression trade-off.

Chapter 4 aims to connect algorithms for ConvNet compression with *hardware awareness*. To effectively deploy ConvNets on edge devices the compression strategies have to be driven by some consciousness of the target hardware. Given a fixed device, with relative hardware constraints, the compression has to meet the requirements still guaranteeing negligible accuracy losses. We first analyzed the memory vs. accuracy solution space to assess the optimality of compressed ConvNets. Then we proposed two novel strategies to compress ConvNets with a budget-aware mechanism: the former under memory constraint, the latter under storage constraint.

Chapter 5 explores the run-time *scalabilty* of ConvNets during inference. Several works have been published to prove the effectiveness to scale ConvNets depending on memory and performance requirements, in particular involving the use of three knobs: depth, width, and resolution. These solutions allow reaching multiple configuration points, with multiple ConvNet models. To overcome this we propose a new training and compression pipeline able to nest a novel class of nested sparse ConvNets with custom training and encoding. Using the sparsity as a dynamic knob, these nested networks are able to adapt latency and accuracy at run-time, yet reducing the memory footprint (storage) and the RAM (memory) needed by inference.

At last, the conclusions and the future research directions are depicted in Chapter 6.

Chapter 2

Background: Overview of ConvNet Compression Techniques

In the last decade, the deep learning community started to deeply work to make the standard ConvNet more efficient without losing original performance. As the standard models are often over-parameterized [84], they bring additional computational and memory costs to the edge device, both for training and inference stages. To overcome this limitation, several techniques have been explored, both novel and inspired by other research fields, with the same objective to improve the ConvNet efficiency and then to facilitate its deployment at the edge. Three of the most popular efficient deep learning techniques are *pruning*, *quantization*, and *encoding*.

As the techniques are proven on different tasks with variable terms of validation, it is not easy to make a clear and comprehensive taxonomy. However, in this chapter, we reported a representative selection of the most popular techniques for efficient deep learning, preceded for the sake of comprehension, by a generic notation summary of the ConvNet inference.

2.1 Generic Notation

Considering a generic ConvNet structure, like the one reported in Figure 1.1, composed by N_L convolutional layers, each l layer is composed by a 4-D weight tensor $\vec{\theta_l} = \theta_l \cdot \vec{x}$. The shape of each $\vec{\theta_l}$ tensor is $\theta_l \in \mathbb{R}^{n \times c \times h \times w}$, where n is the number of input channels (also called filter), c is the number of output channels, while w and h are the width and height in pixels of each 2-D kernel array. During inference, for each 2-D convolution operation ($Conv2D$) over an input signal $\vec{x}$, the generic output tensor $\vec{\phi} \in \mathbb{R}^{c_{out} \times h_{out} \times w_{out}}$ is obtained through Equation 1.1 from the input tensor $\vec{x} \in \mathbb{R}^{c_{in} \times h_{in} \times w_{in}}$, where the triple (c_o, h_o, w_o) corresponds to number of channels, width and height for input or output tensor (if o is equal to *in* or *out*).

The generic $Conv2D$, for a fixed layer, can be described in details with Equation 2.1.

$$\vec{\phi}(c_{out_j}) = \vec{b}(c_{out_j}) + \sum_{k=0}^{c_{in}-1} \vec{\theta}(c_{out_j}, k) * \vec{x}(k) \tag{2.1}$$

It needs to be observed that the number of filters n_F and kernels n_K of the generic tensor $\vec{\theta}$ are respectively equal to c_{out} and c_{in}, while to compute the size of h_{out} and w_{out} is used the Equation 2.2.

$$\begin{cases} hout = \frac{h_{in}+2\cdot pad_h - dil_h\cdot(h_{in}-1)-1}{str_h} + 1 \\ wout = \frac{w_{in}+2\cdot pad_w - dil_w\cdot(w_{in}-1)-1}{str_w} + 1 \end{cases} \tag{2.2}$$

Considering *pad*, *dil* and *str* are the standard image processing operations: *padding*, *dilatation* and *stride*.

2.1.1 Training

Deep Neural Networks are usually trained using back-propagation [169]. This iterative procedure evaluates the derivatives of a loss function L and propagates them backward through the network thus finding the fastest way to reach the minimum value, i.e., the lowest error. At each training step, the network weights, are updated accordingly.

The loss function L is defined as the difference between desired $\vec{\delta}$ and inferred output$\vec{\zeta}$:

$$L(f(\vec{x}, \vec{\theta})) = \left|\left|\vec{\delta} - \vec{\zeta}\right|\right|^2 \tag{2.3}$$

Other formulations of L are reported in [169]. Hence, it is possible to derive a generic cost function $\mathcal{L}$ over the entire training set $\mathcal{S}$ (with cardinality N), given by (2.4).

$$\mathcal{L}(\vec{\theta}) = \frac{1}{N}\sum_{\mathcal{S}} L(f(\vec{x}, \vec{\theta}) = \frac{1}{N}\sum_{\mathcal{S}} \left|\left|\vec{\delta} - \vec{\zeta}\right|\right|^2 \tag{2.4}$$

One of the most popular algorithms used to minimize complex objective functions is Stochastic Gradient Descent (SGD) [10], as Equation 2.4. It guarantees the global (local) minimum for convex (non-convex) functions with reasonable computational efforts. As the name suggests, this method follows the negative slope of the objective function derivatives until a minimum is reached, i.e., the optimal $\vec{\theta}$. At each $(k+1)$-th iteration, the tensor $\vec{\theta}_k$ is updated with decreasing η learning rate as:

$$\vec{\theta}_{k+1} = \vec{\theta}_k - \eta\vec{\nabla}\mathcal{L}(\vec{\theta}_k) \tag{2.5}$$

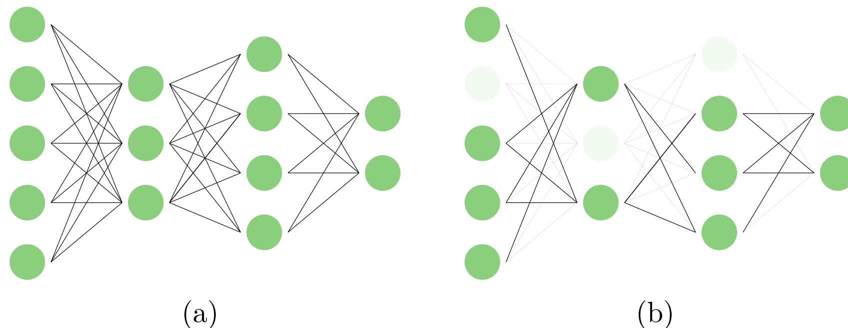
where $\vec{\nabla}\mathcal{L}(\vec{\omega}_k)$ is defined as follows:

$$\vec{\nabla}\mathcal{L}(\vec{\theta}_k) = \frac{1}{N}\sum_{\mathcal{S}} \vec{\nabla}L(\vec{\theta}_k, \vec{x}_k) \tag{2.6}$$

The learning rate parameter η needs a proper assignment to efficiently fine-tune the training algorithm.

2.2 Pruning

Pruning algorithms for generic neural networks models have been firstly proposed by LeCun et al. in [115], before the deep learning popularity. The conceptual pruning idea was to remove unimportant weights from a multi-layer perceptron (MLP) in order to achieve better generalization, decrease the number of training examples required, improve the learning process, and increase the inference speed. This basically means to selectively delete some parts from the model, obtaining a sub-network with equal or different performance. The most popular pictorial scheme about this technique is reported in Figure 2.1, where a generic MLP is reshaped cutting off its redundant connections. Both the network synapses (black lines) and neurons (green circles) can be pruned, operating at different levels of granularity. The main objective is to remove just the unnecessary instances to preserve the knowledge of the model, aspiring to find the optimal trade-off between the number of parameters and prediction performance. Most of the effort is then focused on the optimal selection search. Despite these optimization techniques are heterogeneous and not very recent, in the last years have been largely applied especially on deep convolutional neural networks. Here the structure sizes are bigger and the redundant model parameters number is considerably higher than standard predictive systems.



(a) (b)

Figure 2.1: A generic MLP, before (a) and after pruning (b).

The ideal and optimal solution for pruning a generic neural network would be to empirically search for superfluous parameters iteratively, i.e., by removing one network parameter at a time, usually those that have the lowest impact on accuracy degradation. This procedure should be done for all the layers and for all pruning iterations (N), deleting just one parameter (W) at a time. This “brute-force” approach implies a huge computational complexity ($O(NW^3)$), and it is not

suitable especially for deep networks. In fact, all the main research solutions in this field try to choose a less direct and more approximate approach.

Many solutions have been explored in the last years by several research groups, each one tries to achieve one common goal: adjusting the network model structure to reach better prediction performance with lightweight solutions. However, they do not work in the same way nor are not built for the same reason. On the contrary, they differ considerably between each other with respect to several criteria.

2.2.1 Basic Terminology

Saliency One fundamental concept of pruning algorithms is the saliency, which represents how much the deletion of a certain parameter influences the training error. In general the saliency can be defined as the importance of the parameter weights. In the field of statistical analysis, it is common to easily extract some data information to find an importance order of model parameters. However, this is quite hard in deep learning structures. Exactly for this reason, a saliency measure is often used to sort the network parameters by importance, under the assumption that an instance with low saliency is little effective on the network training error. In Section 2.2.4 we provide a schematic overview of some of the most popular saliency metrics used to prune ConvNets.

Binary Mask Neural Network pruning basically consists in searching and deleting a subset $P \subset \theta$ of parameters, where θ is the original dense 4-D weight tensor. After each pruning application the weight tensor is updated as $\theta^* = \theta \odot M$, where $M \in \{0,1\}$ is a binary mask and $\odot$ is the point-wise product. The mask M is initialized as $M = 1^{|W|}$, and mapped as $M(P) = 0$ during each pruning step: all zero values are placed just on the corresponding positions of subset P. The subset P and the mask M can be updated with different scheduling procedures, as discussed later.

After pruning application, a large amount of parameters is removed from the ConvNet model. In order to validate the effectiveness of pruning there are two popular metrics.

- **Sparsity**: defined as the fraction of zero weights $|\theta^{\emptyset}|$ to the total number of weights:
$$S = \frac{|\theta^{\emptyset}|}{|\theta|} \tag{2.7}$$
Usually, if the number of pruned weights is not negligible, the ConvNet is considered a *sparse model*, the other way around if the ConvNet is considered a *dense model* if there are no zero values. Inducing sparsity inside the network automatically change the distribution of weights, as showed in Figure 2.2.

Usually, over 50% of the parameters are pruned from the original ConvNet to enable the use of efficient sparse storage formats [84].

- **Compression**: usually defined as the ratio between the sizes of the pruned model respect the original one:

$$CR = \frac{\theta \odot M}{\theta} \tag{2.8}$$

This metric is mostly used when pruning change the original topology of the ConvNet, for example, if entire filters are removed from the structure (Section 2.2.2). The expected compression factor strongly depends on the ConvNet architecture and the hardware constraints, however, it is common to compress the model at least 2× with respect to the original one [135, 131].

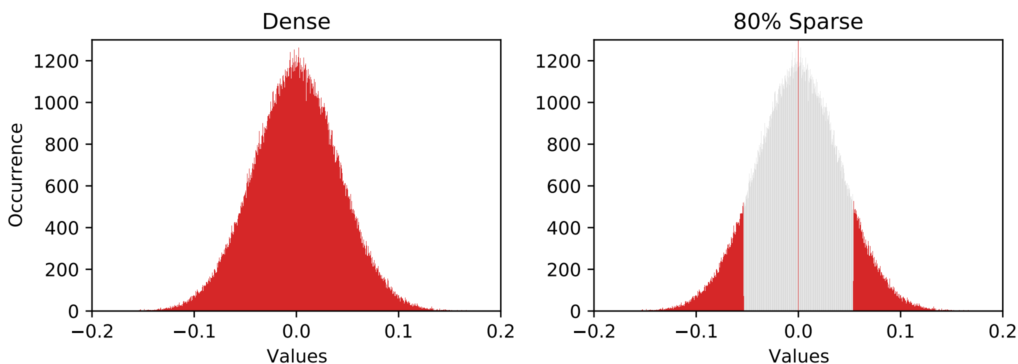
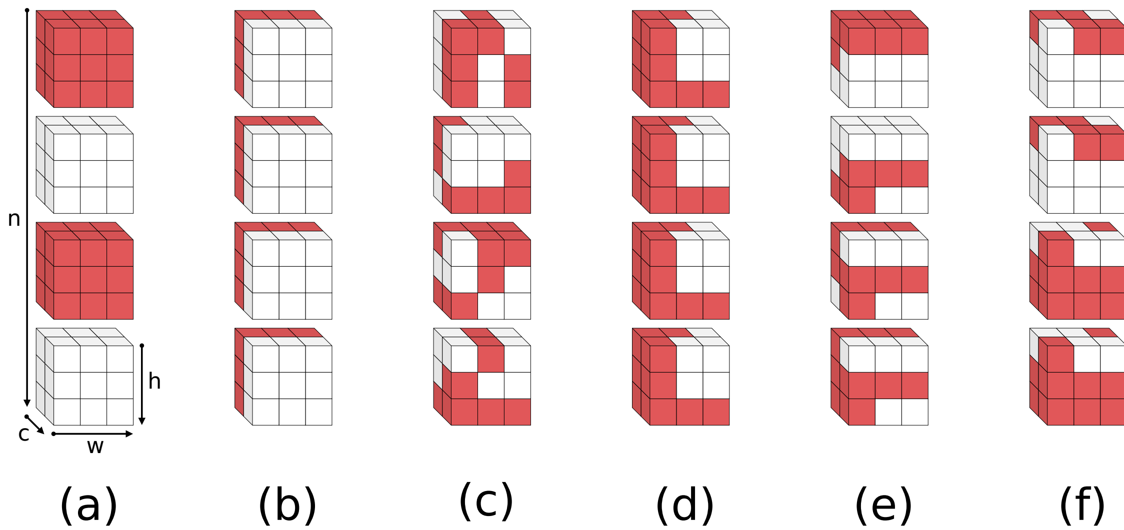


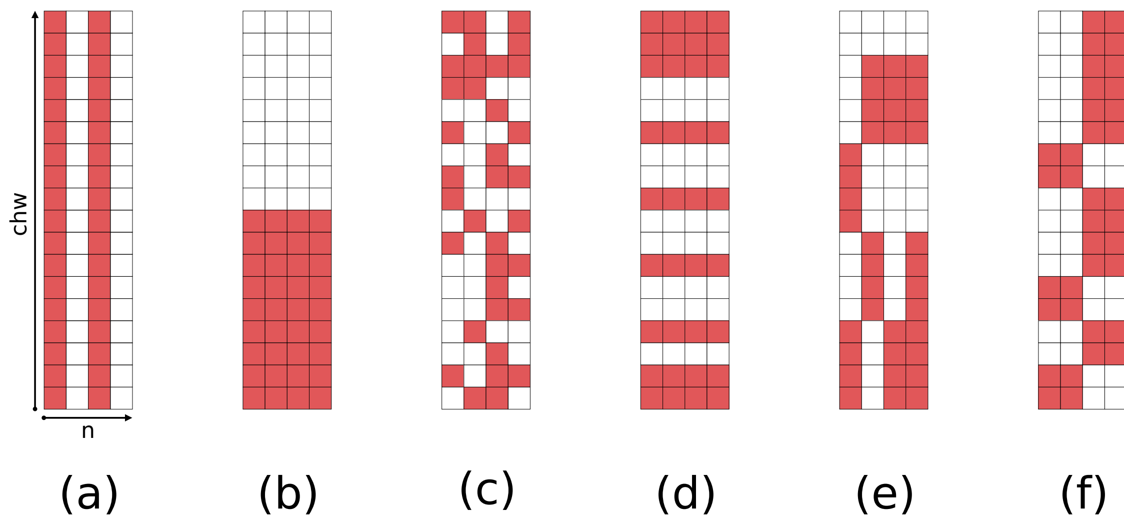
Figure 2.2: The figures show how pruning influences the distribution of the weights, before (on the left side) and after (on the right) its application. The example refers to a magnitude-based pruning to the last convolutional layer of MobileNet_v2 architecture trained on Imagenet (with 320 input channels and 960 output channels). The 80% of the weights of the layer are pruned. On the right side, the figure the y-axis has been limited for the sake of comprehension, in order to avoid the huge peak representation in 0 (histogram value = 245760).

2.2.2 What to Prune

An essential aspect of a pruning algorithm is its granularity, i.e., the hierarchical level of the network structure where the algorithm is applied. Several solutions have been explored in the last few years, but all of them can be grouped in two main categories: *unstructured* or *structured*. While the first basically induce sparsity inside the model removing "sparse" weights, the second removes entire *structures* from the model (i.e., grouped weights, filters, channels, etc.). We present now a more detailed comparison between these two categories. Two pictorial schemes of the most common pruning granularities are reported in Figures 2.3a, 2.3b, according to different projection views.



(a) *(n,c,h,w)* layout.



(b) *n-chw* layout.

Figure 2.3: Most popular levels of pruning granularity, represented in an example layer of shape $\mathbb{R}^{4\times2\times3\times3}$. The *structured* levels are *a* and *b*, while the *unstructured* levels are *c, d, e, f.* The pruned parameters are colored in red.

Unstructured Pruning

This branch of pruning techniques aims to reduce the number of nonzero parameters inside the model, increasing the numerator of the Equation 2.7. This optimization does not modify the network topology, but just the values inside its

tensors. Indeed it does not bring immediate gaining in terms of model efficiency, as the model structure remains the same. However, the memory footprint can dramatically be reduced if the *unstructured* pruning is combined with sparse-aware encoding strategies, as described in Section 2.4. While this procedure requires proper encoding and ad-hoc kernel libraries [38] to optimize memory, storage, and latency on general-purpose architectures, this procedure has been the most explored in literature, especially in more recent works, as it allows to have best specific control on the network parameters and also it was already used on standard neural networks. They are identified as (c,d,e,f) in Figures 2.3a, 2.3b. Some of the most popular pruning works at unstructured granularity are proposed in [72, 61, 145, 239, 225, 44, 134].

Weight This is the lowest granularity of the network structure, pruning operates directly on the redundant weights. This is also called fine-grained pruning, as it works exactly on the individual weights. The direct efficiency of this granularity approach is the lowest. During each 2-D convolution operation over an input signal $\vec{x}$, for each pruned parameter, the operations saved are equal to $c_{in} \cdot h_{in} \cdot w_{in}$.

Block Another solution is to group close weights in some blocks (or patterns) of various shapes. In this way, the procedure aims to induce a structured sparsity inside each model, according to a fixed pattern. The block shape is generally defined as $bs \in \mathbb{R}^{n \times m}$, where n and m are the height and width of the pattern. The blocks usually are considered as unique and non-overlapping instances composed of chunks of individual neighboring weights. We reported three popular examples block-pruning in Figures 2.3a, 2.3b. Considering a block composed by s weights, the block can be chosen in direction of output channel $s \times 1$ (d), or input channel $1 \times s$ (e), or instead without any direction $\frac{s}{2} \times \frac{s}{2}$ (f). The possible shape can be chosen according to the custom architecture of the kernel library with the aim to take advantage of data reuse for inference speed-up. Respect single-weight pruning, this approach is certainly more effective in terms of inference latency: bigger block-size brings higher inference speed, at the expense of a loss of control. For each pruned block, the operations saved are equal to $s \cdot c_{in} \cdot h_{in} \cdot w_{in}$.

Structured Pruning

This category contains all the strategies that remove entire structures from the original model, like neurons (for original MLP), channels, or filters (for ConvNets). Contrarily to unstructured strategies, using these solutions the model is directly compressed, as it basically changes its architecture. It needs to be observed that for each pruned structure (channel or filter) inside a layer l, it is automatically removed the connected structure in the following layer $l + 1$. In this way, the

feature maps between the two layers of the ConvNet are implicitly reduced. For these reasons, the *structured* granularity is the most efficient for the model, but it is also the most difficult to handle because it is quite rare to have an entire redundant filter without any useful weight, whereas it is more common to have just a subset of its internal parameters that need to be pruned. They are identified as (a, b) in Figures 2.3a, 2.3a. Some of the most popular pruning works at structured granularity are proposed in [119, 147, 5, 131, 80, 135, 182, 86].

Channel It focuses on the deletion of the second dimension of the tensor $W \in \mathbb{R}^{n \times c \times h \times w}$, it is sometimes called *kernel* as it removes the 2-D slice matrix of each 3-D filter, as reported in Figure 2.3a-b. The channels are considered as unique instances to prune according to a certain score, which is normalized between all the weights inside it. This approach allows to reach high compression rates: for each pruned channel the number of operation saved in the layer l is equal to $n \cdot w \cdot h \cdot c_{in} \cdot h_{in} \cdot w_{in}$. Remembering that n is the number of input channels or filters, the operations saved can be simplified as $whc_{in}^2 w_{in} h_{in}$.

Filter The filter level approach is quite similar to the previous one, with the main difference that now the pruned instances are the filters n, or input channel (according to the nomenclature reported above). It removes entire 3-D filters inside each layer, hence focusing on 3-D structured sparsity (Figure 2.3a-a). For each pruned filter in the layer l the algorithm saves $c_{out} \cdot w \cdot h \cdot c_{in} \cdot h_{in} \cdot w_{in}$ operations. Filter pruning can be considered as the corresponding of *neuron* pruning for Multi-Layer Perceptrons (MLP).

2.2.3 When to Prune

The broad aim of the pruning algorithms is to delete all the redundant parameters from a network model without performance loss, to find the best efficient accurate model for that fixed objective. To accomplish this task, the most common pruning flow includes three stages: (i) *pre-training*, (ii) *pruning* and (iii) *fine-tuning*. The pre-training is the standard dense training to reach some starting accuracy state; the *pruning* phase comprises the actual *masking* of the model with the relative update of the binary mask; at last, the *fine-tuning* allows the model to freeze the masks and recover from the loss of prediction accuracy, due to the new limited data space. The actual position of these three stages can be scheduled differently in the training flow pipeline.

The most popular schemes to schedule pruning inside a full training procedure are two: *prune once and retrain* and *iterative pruning and retrain*. The following text briefly describes both these pruning strategies, however, for sake of simplicity, they are both presented for the unstructured pruning case, where the ConvNet is

sparsified without any topology variation. It needs to be noted that this is not a loss of generality, as the same algorithms can be easily extended also for structured pruning cases.

- **Prune once and retrain** (Algorithm 1). At first, the model θ and the binary mask M are initialized (line 1). Then the ConvNet is trained for a fixed number of pre-training epochs $epochs_{pre}$, obtaining a dense trained model (lines $2-4$). Hence a mask M is extracted using a $get_mask(\cdot)$ function (line 5), which basically sorts and selects the parameters θ according some importance criteria. The quantity of parameters to prune is fixed by the sparsity S. Then, the dense model is multiplied with the mask to obtain the sparse model θ_s (line 6). At last, there is a fine-tuning stage for a fixed number of post-pruning epochs $epochs_{post}$ (lines $7-9$), to restore the original accuracy still preserving the pruned weights in the fixed position. In this fine-tuning stage, the gradient can flow just on the dense weights to update them, while the sparse weights do not receive the gradient as they cannot be updated. In this way, the sparse ConvNet can re-adapt the dense parameters to the constrained solution space.

- **Iterative pruning and retrain** (Algorithm 2). At first, the model θ and the binary mask M are initialized (line 1). Then the model is pruned and retrained iteratively for a fixed number of epochs (lines $2-8$). Differently with respect to the previous strategy (2.2.3), here the procedure is driven by two schedulers: the $frequency_scheduler(\cdot)$, which decides when to prune or not prune, and the $sparsity_scheduler(\cdot)$, which decides how much to prune. For each e-th epoch, after the standard train step (line 3), the $frequency_scheduler(\cdot)$ checks if the model has to be pruned (line 4). If yes, the $sparsity_scheduler(\cdot)$ extracts the percentage of parameters to prune S (line 5), then it updates the mask M (line 6) to generate the sparse model θ_s (line 7). The $get_mask(\cdot)$ function works the same way as before (2.2.3), with the main difference that here the function is called every pruning epoch: the purpose is to update the mask M multiple times to better learn the correct locations of redundant weights. This is the most popular scheduling approach in the literature, as it allows the model a smoother and softer adaption to the induced sparsity, with no brute force changes which may be harder to tolerate.

Frequency and Sparsity Schedulers

As introduced in the previous section, two are the main schedulers who drive the iterative pruning and training loop: the *frequency scheduler*, which answers to the question "*when to prune*", and the *sparsity scheduler*, which answers to the question

Algorithm 1: Prune once and retrain.

```
  Input: θ, epochs_pre, epochs_post, S
1 Model Init: θ ← θ_0, M = 1^|W|
2 for e in epochs_pre do
3 |  train_step(θ)
4 |  test_step(θ)
5 M ← get_mask(θ, S) θ_s ← θ ⊙ M // Prune
6 for e in epochs_post do
7 |  train_step(θ_s)
8 |  test_step(θ_s)
9 return θ_s
```

Algorithm 2: Iterative pruning and training.

```
  Input: θ, epochs, frequency_scheduler, sparsity_scheduler
1 Model Init: θ ← θ_0, M = 1^|W|
2 for e in epochs do
3 |  train_step(θ)
4 |  if frequency_scheduler(e) then
5 |  |  S ← sparsity_scheduler(e) // Get Sparsity
6 |  |  M ← get_mask(θ, S) θ_s ← θ ⊙ M // Prune
7 |  test_step(θ_s)
8 return θ_s
```

"*how much to prune*". They both operate discretely inside the training loop, being called at each training step t (forward pass of one single batch of samples) or at each training epoch e (forward pass of all the batches of the training data, full cycle), the technical difference is just an implementation choice. However, to align our description to the most popular works on this topic, the following text introduces these two scheduling strategies in function of the training step t.

The *frequency scheduler* gets in input the training step t and provides a $True$ flag if the t-th step is a pruning step, $False$ if it is not. During the full training stage, this schema prunes the model starting from a t_0 step and ending at t_f step, with a Δt frequency. In this way the full set of pruning steps is defined as $t \in \{t_0, t_0 + \Delta t, ..., t_0 + n\Delta t\}$, where n is a fixed hyperparameter.

The *sparsity scheduler* instead indicates, each time is called, the fraction of parameters to prune. They can be mostly of two types.

- **Constant**. The scheduler simply provides the target sparsity value s_t, as a unique constant value through all the sparse training processes. However, it is still zero if the step belong to pre-training stage $t \notin [t_o, t_f]$.

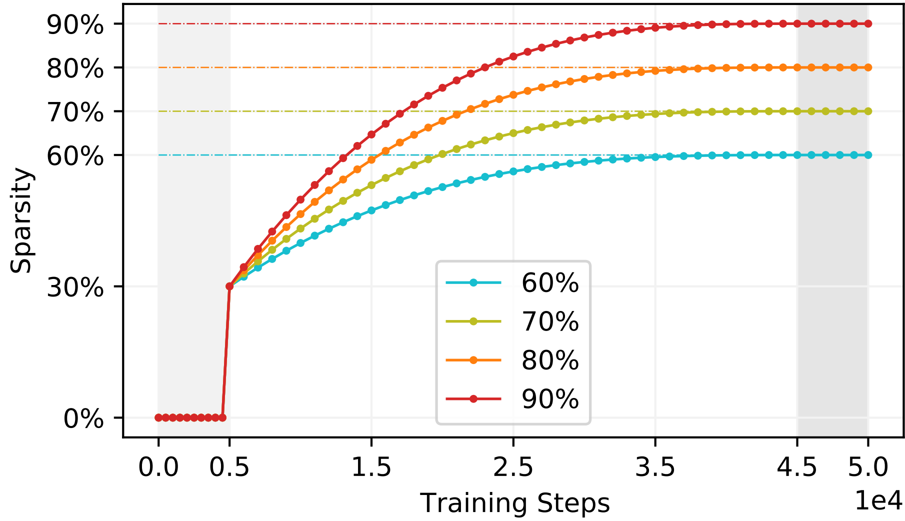


Figure 2.4: The polynomial decay sparsity scheduler. The results are shown for 4 different sparsity levels $s_f \in \{60\%, 70\%, 80\%, 90\%\}$, with same scheduler setting: t_0=5e3, s_i=30%, p=3. Each dot marker represents a pruning step, the frequency was fixed at 1000. The light-grey area (on the left) represents the dense pre-training stage, while the dark-gray area (on the right) indicates the mask freeze stage where the sparsity stops to increase.

- **Monotone Increase**. The sparsity increases monotonically during the training process. The scheduler provides, for each pruning step t, a different sparsity value s_t following a predefined monotonic function such that $t_x \leq t_y$ gives $s_{t_x} \leq s_{t_y}$. The most popular example of this type of sparsity scheduling has been introduced in [239], where the authors proposed a *polynomial decay* function to modulate the increase of the sparsity, in a generic and effective way. The proposed decay function is reported in Equation 2.9, where p is the power of the polynomial decay, s_i and s_f are the initial and target sparsity values, and t, t_0, $n\Delta t$ are defined by the frequency scheduler.

$$s_t = s_f + (s_i - s_f)\left(1 - \frac{t - t_0}{n\Delta t}\right)^p \tag{2.9}$$

It needs to be noted that the power p determines the shape of the monotonic function, in fact, despite it is usually fixed at 3 for standard polynomial decay, it can be fixed to $p = 1$ to generate a linear function, or to $p = 0$ to generate a constant function. For sake of comprehension, Figure 2.4 provides an example of the sparsity scheduling, where the trend of sparsity increase is shown for different target values $s_f \in \{60\%, 70\%, 80\%, 90\%\}$. This shows how the sparsity percentage gradually increases from s_i to s_f, trying to prune rapidly in the first epochs when the redundant connections are abundant, then slowing

down the sparsity rise when the model balance becomes more fragile. Despite this scheduler was originally proposed for unstructured pruning, hence working on the sparsity, it can be applied also on structured granularity levels.

2.2.4 How to Prune

Selecting the Candidates

Defining the correct saliency is the real core of any pruning algorithm, as it selects the parameters to delete. The most intuitive approach, and one of the first explored, is to evaluate the ConvNet with and without a certain set of parameters [183]. However, the leave-some-out approach is not trivial especially for larger ConvNets, as it requires multiple training processes, one for each subset of parameters to evaluate. More recently, several works assessed the efficiency of a random selection of pruning candidates [144, 17], leaving the job to recover the accuracy loss just to the ConvNet plasticity during the fine-tuning stage. This strategy, partially related to the compressive sensing theory, has been proven to be effective in some particular settings.

However, despite there is no clear understanding of which is the most effective strategy to select the pruning candidates under all the possible boundary conditions, several works demonstrated that using some saliency metric is useful to drive the procedure of candidate selection [46]. Pruning is a very explored topic in efficient deep learning research: many different strategies have been proposed to better accomplish the removal of redundancy from the ConvNet. Considering the elevated number of different methods, with sensitive differences of settings (i.e., scheduling, tasks, hyperparameters, etc.), a clear and exhaustive taxonomy of the mechanism with a fair quantitative comparison of the effectiveness is very challenging. However there are some works that tried to overview the strategies [84, 46], and others that compared the different techniques in controlled conditions (i.e., with fixed setting) [8].

In this section we provide a brief introduction to some of the most popular strategies to drive the selection of the candidates to prune.

Magnitude ConvNets can be pruned just on the basis of their parameters. This strategy is a *data-free* selection, as during the selection of the candidates it avoids any relation with the samples. One of the simplest, but also most effective, approaches is the magnitude-based selection: it removes model parameters according to their absolute magnitude. Various works have been published following this concept, from the oldest [66], to the more recent and popular [67, 47, 239]. The intuition was simple: lower magnitude means lower significance. To compute the importance score to generate the pruning masks, usually, the saliency is computed according to $\sum_G |w_i|$, where the sum is computed for all the weights w inside a

group G (i.e., a filer, a channel, a block, or a single weight for fine-grained pruning). Then the ConvNet is pruned according to $\sum_G |w_i| \leq x$, where x is a threshold dependent by the sparsity. The magnitude-based pruning has been applied heterogeneously on unstructured and structured granularity. A very popular example of the former is proposed in [67] pruning weights for a deep compression pipeline of ConvNets, while a popular example of the latter is proposed in [119] focusing on removing whole convolutional filters with the small absolute norm for efficient ConvNets. Furthermore, the same pruning strategies have been also explored effectively on recurrent neural networks in [150, 174]. Other works, used criteria based on the similarity of the parameters to prune networks. It has been shown in [182] that this approach allows merging similar neurons together with effective results, especially on small size ConvNets.

Data-driven ConvNets can be pruned also considering the sensitivity of the model to the training data. The key concept is that elements that do not change respect the deviation of the input samples are redundant since their output is almost constant to the input changing. Hence model parameters are pruned according to some measure of this *sensitivity*. Some prior works about this strategy have been proposed in [179, 15]. More recently the same approach has been applied to prune convolutional filters in [135] according to how much they influences the following layers. Other popular approaches are proposed in [226, 35, 79].

Taylor Expansion of the Training Loss Loss functions play a central role in machine learning algorithm mapping decisions to their associated costs. Therefore, it is realistic that estimating the impact of weights perturbation over the loss function represents a simple yet efficient mechanism to control the pruning procedure. However, it would be prohibitively laborious to evaluate the loss function variations directly from this definition, i.e., by deleting each parameter and re-evaluating the loss. For this reason, the algorithms of this class of pruning leverage local models of the training loss function and analytically predict the effect of perturbing the parameters.

In the literature, the Taylor expansion of the training loss L is one of the most efficient analytical tools to estimate local loss variations. During the training process, network's parameters $\vec{\theta}$ are optimized such that the loss function is minimized. Hence, the pruning process tries to find the best θ^* generic set, with a subset of deleted parameters, that minimize the L function change $|\Delta L| = |L(f(x;\theta^*),\zeta^*) - L(f(x;\theta),\zeta)|$. In order to solve this optimization problem, with a reasonable computational load, the loss L change can be rewritten, using the Taylor expansion being W a generic parameter set as in Equation 2.10.

$$\Delta L = \frac{\partial L}{\partial \theta}\Delta\theta + \frac{1}{2}\frac{\partial^2 L}{\partial \theta^2}\Delta\theta^2 + ... \tag{2.10}$$

Usually, just the first two terms of the Taylor expansion are considered, as the higher order ones are negligible and then ignored.

There are two sub-branches that use the Taylor expansion to drive the pruning process.

The first branch includes the so-called *gradient-based* pruning techniques, which directly use just the first term of the expansion. The intuition is to select and remove the weights with small variations during the learning process, focusing on the absolute value of the gradient. These strategies assume to avoid the computation of the second derivative term of the Taylor expansion, as they assume that the first term tends to zero after the training completion when in the local minimum the gradient term tends to zero $\frac{\partial L}{\partial \theta} \rightarrow 0$ [147]. However, its variance is a non-zero term, and it is possible to use it as a pruning criterion as it contains information regarding the stability of the cost function. Hence they use the expected value (proportional to its variance) of the first-order term of the Taylor series as a pruning control metric because it is empirically more informative and it allows to speed up the algorithm computation. They use the absolute value of the gradient to determine whether a parameter should be removed or no Examples of the use of the first term of Taylor expansion can be found in [146, 147, 118, 214, 36]. A similar approach has been applied in a recent work presented in [173], where the authors focused on the direction of the gradient during the learning process to prune transformer-based models.

Involving the second term of the Taylor expansion has been largely explored in literature. Pioneering works on pruning drove the selection with this procedure: *Optimal Brain Damage* (OBD) [115] is an example. The authors estimated the saliency of the parameters by evaluating the impact of their variation on the training error; *Optimal Brain Surgeon* (OBS) [75] extended OBD concept with a generalization of the model to compute the error increase due to some weight elimination. They are both based on the Hessian matrix H (or *Hessian*) of a function $f : \mathbb{R}_\mathrm{n} \rightarrow \mathbb{R}$, which is defined as a square $\mathrm{n} \times \mathrm{n}$ matrix of the f second-order partial derivatives $h_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$, if they do exist and are continuous over the domain of the function. The Hessian is usually employed to describe the local curvature of a n-variables function: the higher the values of $h_{i,j}$, the steeper the curvature of the function. Even though this metric was employed in the earliest pruning techniques, specifically designed for MLPs, their functionalities represent the basis of more recent approaches. Other more recent works based on these concepts are proposed in [37, 191, 203, 69]. These approaches have strong mathematical frameworks for pruning, however, they are based on several assumptions that make their use not always easy.

Regularization of the Training Loss A high number of works includes the model pruning directly on the regularization process. The concept is to add a penalty term $P(\vec{\theta})$ inside the cost function $\mathcal{L}(\vec{\theta}) = \mathcal{L}(\vec{\theta}) + P(\vec{\theta})$. Several penalty

functions to drive the search of the redundant parameters to prune [242, 145, 220, 26, 126, 227]. The complexity of this type of pruning strategy is all enclosed in the definition of $P(\vec{\theta})$, as the resulting problem is often non-convex and quite challenging to solve (it is common to have additional local minima [73]). Furthermore, additional parameters have to be included in the training loop, increasing the order of complexity of the hyperparameters optimization.

Variance At last, ConvNet pruning can be faced as a real information theory problem, searching the redundant elements according to their distribution. Higher variance means a lower contribution to the inference. Given a training set $\mathcal{S}$ composed by input data x and predictions y, with relative model parameters θ, the Bayesian Learning theory considers a prior knowledge $p(\theta)$ of the distribution of the weight. The *Bayesian Inference* process is defined as the posterior distribution extraction, from the prior distribution defined as in Equation 2.11.

$$p(\theta|\mathcal{S}) = p(\mathcal{S}|\theta)p(\theta)/p(\mathcal{S}) \tag{2.11}$$

The Sparse Variational Dropout presented in [145], extended the Variational Dropout method [107] in the interest to induces sparsity inside the ConvNets. Then, a more theoretical approach has been explored in [133], where a Bayesian compression pipeline is adopted to prune large parts of the network through priors that induce sparsity. The technique aims to generalize the Variational Dropout, focusing on higher levels of granularity (filters). Other popular works using variance as a guide to optimize ConvNets are proposed in [196, 151, 163]. The main benefits of these strategies are the absence of the hyperparameters to tune, and the avoidance of the fine-tuning stage; in fact at the end of the training the parameters with larger dropout values can be cut-off from the model in one shot. However, they need to double the parameters of the model, and their training from scratch is quite challenging [47, 84].

Homogeneity

At last, it needs to be observed that pruning can be applied with different homogeneity across the ConvNet layers. This means that the full model can be pruned with uniform sparsity (or pruning rate), or instead, it can be modulated by some layer-wise scoring.

- Global-wise. This is the most popular approach and it provides the same uniform amount of pruned parameters across all the layers. However, it is common to avoid pruning the first (and sometimes the last) layer, to avoid huge accuracy losses.

- Layer-wise. This approach aims to find the optimal rate of sparsity inside each layer, using some heuristic or optimization tool. For example, Variational Dropout [145] used a Bayesian technique to address a non-uniform sparsity across the layers, with no manual intervention. This could be more effective in very deep models, where the last layers tend to have less significant information. However discriminating which layers prune more is very challenging, as the risk to prune important weights is high, generating pruned models less accurate than ones optimized with the global-wise approach.

2.3 Quantization

One of the simplest ways to reduce the complexity of a ConvNet is to reduce the numerical precision of its parameters and activations. This approach is called quantization. The first feature to highlight is the versatility of this technique, which can be applied equally across different models and use cases, as it does not require generating a different model from the original one to obtain a more efficient solution. In fact, quantization can be applied directly on the parameters of the full precision model (both weights and activations), usually from 32bit floating-point to fixed-point precision.

Figure 2.5 shows a generic example of quantizaton applied on the first convolutional layer of MobileNet_v2 model. The left picture shows the weight distribution with full precision (32bit floating point), while the right picture shows the weight quantized to 8bit. It is clear to see how the distribution shape still remains the same, but the number of levels and the occurrence change sensitively.

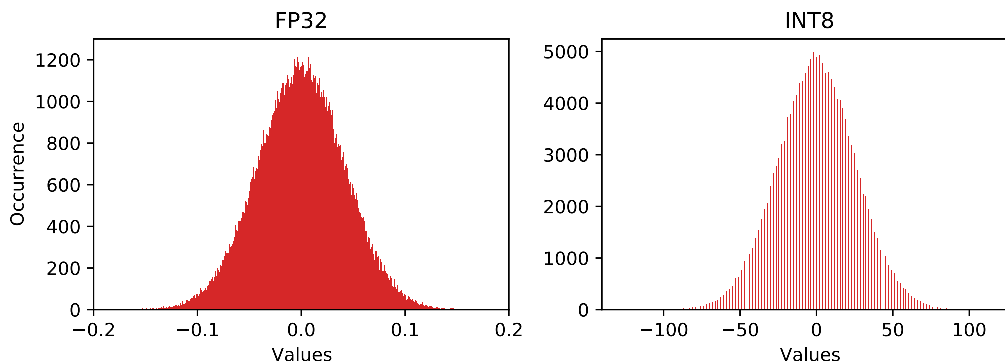


Figure 2.5: Before (left) and after (right) weight quantization. The example refers to the last convolutional layer of MobileNet_v2 architecture trained on Imagenet (with 320 input channels and 960 output channels).

The main advantages provided by quantization can be summarized as follows:

- **Memory Footprint**. Reducing the numerical precision means directly reduce by a N factor the memory footprint of the original model, where N is the ratio between original and quantized precision. For example, with only 8-bit quantization, the memory footprint is reduced by a factor of $4\times$.
- **Memory Bandwidth**. The memory required to store intermediate computations is automatically reduced when the data used to compute matrix multiplication are with reduced numerical precision.
- **Speed**. The inference computation can speed up due to memory bandwidth savings and also because many processors work faster with int8 arithmetic.

- **Power**. Moving reduced precision data is more efficient as in many architectures the memory accesses can dominate the power consumption. For example, move 8-bit of data may require 4× less time than moving the same data stored in 32-bit.

The quantization can be applied just on the weights or also on the activations. If it is applied only on the weights the main benefits are on the memory footprint reduction, while if applied also on the activations the quantization allows also a sensitive inference speed up. Quantize also the activations allows higher processing efficiency to reduce the inference time. In fact, during the inference stage, the $int8$ weights are multiplied to $int8$ activations, which are generated on fly just before the computation (this is the reason of the *dynamic* name).

A generic scheme of quantization is the one proposed in [98]. Considering a generic floating point variable $r \in [r_{min}, r_{max}]$ which needs to be quantized to an integer $q_{int8} \in [0, N-1]$ where N is the number of levels (for $int8$ $N = 256$). The affine mapping is defined with the quantization scheme

$$r = S(q_{uint8} - z) \tag{2.12}$$

where S and z are the quantization parameters. S is the *scale*, a real positive number who specifies the step size of the quantization; z is the *zero-point*, an integer who represents the quantized value q corresponding to the real value 0, useful to ensure common boundary operations like zero padding. Then the quantization procedure is then defined as

$$q_{int8} = round(\frac{r}{S} + z) \tag{2.13}$$

$$q_{uint8} = clamp(0, N-1, q_{int8}) \tag{2.14}$$

where $q_{int8} \in [-\frac{N}{2}, \frac{N}{2}]$ is the unsigned integer value shifted of $N/2$ from q_{int8} with the $clamp(\cdot)$ operation 2.15.

$$clamp(a, b, x) = \begin{cases} a & \text{if } x \leq a \\ b & \text{if } a \leq x \leq b \\ x & \text{if } x \geq b \end{cases} \tag{2.15}$$

This scheme is also defined as *asymmetric* quantization as the distribution is shifted by a given offset z respect the zero value.

A particular case is the *symmetric* case, where the distribution is centered around zero, as the zero-point z is equal to zero. In general, this solution is simpler to implement but less accurate compared to the asymmetric case, which, however, encompasses additional processing stages [112].

$$q_{int8} = round(\frac{r}{S}) \tag{2.16}$$

The de-quantization operaations is now simplified as

$$r = S(q_{uint8}). \tag{2.17}$$

Linear vs. Non-Linear

The weights can be quantized *linearly* or *non-linearly*. A *linear* approach [93] applies a uniform distance between all the quantized weights; this is the simplest solution but yet the most suitable for general-purpose hardware, in fact, it requires a lightweight implementation. Higher compression can be obtained with *non-linear* quantization, this solution applies a particular function for mapping the full precision parameters into a discrete fixed-point space. It ensures more accurate profiling of the original distribution, usually not uniform, guaranteeing lower accuracy losses with higher compression rates. The most popular examples are the *log-domain* [117] and *clustering* approaches [67] like k-means. However, it needs to be observed that non-linear schemes require the use of hashing functions, which are commonly implemented by custom hardware to improve performance [202] or, alternatively, by software routines whose severe overhead affects latency.

Granularity

The quantization operation can be applied at different granularity levels: tensor, layer, and channel. This means that the quantizer operation with the same settings and rules (i.e. bit-width, zero-point, scale, ...) can be defined as *fixed* when equally applied to all the parameters of the ConvNet, or *variable* when applied differently according to some substructures (layer or channel) [149]. Operating at lower granularities sensitively helps to improve the final accuracy as showed in [162]. For example, adapting the quantization at the channel level means that each convolutional kernel has a different custom setting respect the others. Hence, *variable* quantization is more accurate at cost of higher complexity both for conversion and inference computation.

Radix Point

Finally, it is possible to quantize the weights using a *binary radix-point scaling*, or an *arbitrary linear scaling*. The former uses a simple bit-shift operation for scaling among the quantized levels [112], the latter requires additional operations. Then the *binary radix-point* scaling is simpler and less accurate, while *arbitrary linear* scaling is more accurate but slower as it increases the global latency [112].

2.3.1 Quantized Inference

There are two main approaches to quantize a ConvNet model for the inference process: *post-training quantization* and *quantization-aware training*. At the end of the process, both of them bring the model in a quantized format ready to be deployed. A generic 8-bit quantized inference scheme for a convolutional layer is

shown in Figure 2.6. Both weights and input/output activations are in unsigned-integer format (*uint*8) according to Equation 2.12, while biases are kept in unsigned-integer at 32bit (*uint*32). Convolution uses 8-bit arithmetic operands and 32-bit accumulator, *ReLu*6 uses 8-bit integer arithmetic, while bias addition only 32-bit integer.

Post-Training Quantization

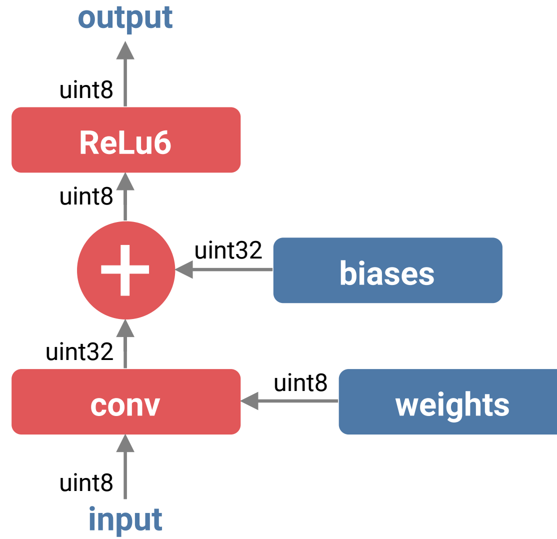


Figure 2.6: Quantized inference (uint8).

The easiest way to quantize a ConvNet is to compress the weights and eventually the activations after the training process. This approach does not insert the quantizer operator inside the training loop but just applies the quantization once the weights are fully trained. If the process is just applied on weights the benefit is fully on the memory footprint reduction. If the process involves weights and activations the quantization enhances all the benefits cited above, but it requires a further stage of calibration to compute the dynamic ranges of activations. As showed in [108] usually most of the accuracy loss is due to weight quantization, as activations have not high negative influence.

The authors of [108] showed that using asymmetric post-training quantization of both weights and activations at int8 is possible to reach close to the floating-point accuracy for a large set of ConvNets. Usually, most large ConvNets (like ResNets and Inception-v3) are more robust to weight quantization compared to thinner models like MobileNets.

Quantize weights at layer granularity can influence sensitively the accuracy, especially for the smaller models like MobileNets. Activation is not affected by this

problem. This phenomenon is mainly due to batch normalization who generates high variations in a dynamic range of the convolutional channels inside each layer. On the other hand, use a per-layer quantization scheme to overcome this issue, as the accuracy becomes independent from the batch-norm scaling/ A possible approach to improve accuracy loss due to batch-normalization is weight regularization as showed in [177].

Quantization-Aware Training

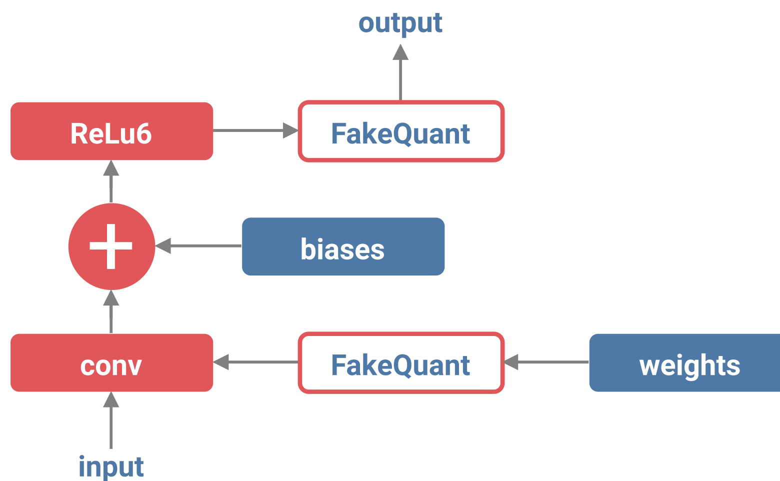


Figure 2.7: Quantization-aware training

A different quantization approach to include the numerical precision approximation inside the training loop is the quantization-aware training. This scheme provides higher accuracy especially for small-size models, at cost of higher implementation complexity.

Usually, this approach is modelized with a *fake-quantization* operator to insert in the training graph, which keeps the weights and activations in full precision floating point during backpropagation passes, while quantize them during each forward propagation simulating what happens in a real inference engine. In detail, for each training step, the weights are quantized before to be convolved with input, while the activations are quantized at the same point of real inference that is after the convolution operation (or linear operation for fully connected layers, or after a bypass residual connector). During backpropagation instead, the weights are kept floating-point to accumulate the gradient at maximum precision for the updating. This ensures avoiding underflowing some small updates for the quantization. A comprehensive explanation of the quantization-aware training is shown in [98] A

pictorial example of the working of fake quantization ($FakeQuant$) in the convolutional layer is represented in Figure 2.7. All the process involves 32-bit floating-point arithmetic (both variables and computations), the FakeQuant operand simulates the effect of quantization during the training step both for weights and for the output activations.

This quantization scheme is not trivial to implement, however, it allows to close the gap to floating-point accuracy, also for layer-wise granularity. Furthermore, the authors of [143, 108] showed that start the quantization-aware training from a preliminary floating-point checkpoint is better than starting to quantize the weights from scratch, as the final accuracy is always higher. This phenomenon can be explained considering that a generic ConvNet can reach higher accuracy when it learns from a teacher model trained with higher degrees of freedom [83].

2.3.2 Quantize for the Edge

The first works on ConvNet quantization demonstrated that 32-bit Floating-Point ConvNets can be quantized to 16-bit and 8-bit Fixed-Point [165] still ensuring negligible accuracy loss. In these cases, the quantization provides a memory compression proportional to the reduction of the bit-width and negligible accuracy loss especially for over-parametrized ConvNets, like AlexNet, VGG, ResNet [165]. Then, more extreme optimization approaches tried to increase the compression going deeper with the quantization precision up to ternary [4] or binary [168] representations. Such an extreme level of bit-width reduction allows a linear compression of the memory footprint, but mostly, it improves the memory bandwidth as multiple operands can be written/read within single access.

Unfortunately, general-purpose MCUs support a discrete set of integer options, e.g. 16-bit and 8-bit for the Cortex-M. Going deeper with quantization is not straightforward: bit-width smaller than 8-bit (e.g. from 7- to 2-bit [195]) remain a theoretic study as they need custom hardware components that are not available in low-power cores (e.g. variable bit-width integer MAC units and/or special memory architectures).

Some IoT platforms offer the 4-bit, e.g. the GAP-8 powered by the PULP core [27]. For instance, a 32-bit SRAM line can host four 8-bit weights that can be easily fed to the execution units, while the use of 9-bit weights incurs in memory under-utilization and it requires specialized unpacking routines that affect latency [170]. ConvNet accelerators with arbitrary bit-width arithmetic, e.g. [210] [176] [197] (FPGA-based), are options. However, they dissipate more power than the MCUs showed in Table 1.1, ($\geq$ 300mW vs. tens of mW).

Not least, quantization below the 8-bit mark (e.g. from 7- to 2-bit [195]) remains a theoretic study as it asks for custom hardware components that are not available in low-power cores, e.g. variable bit-width integer MAC units and/or special memory architectures. However, there are some low-power IoT cores able

to offer 4-bit instructions, e.g. the GAP8 [49] powered by the PULP core [27], but up to now with no ready-to-use IoT solutions for the arbitrary bit-width scaling. Moreover, there are specialized neural cores for multi-bit resolutions, like the Imagination Series 2NX [210], but yet with a power budget of few Watts. Other custom solutions, programmable [197] or hard-wired [176], are not very suitable for the IoT budget cost. For general-purpose architectures, the storage of weights with arbitrary bit widths needs the use of specialized memory allocation strategies to store multiple operands in a single memory word. However, this may bring high-performance degradation due to additional operations to unpack the operands and feed regularly the execution units, as shown in [170]. Then, also solutions based on programmable devices (e.g. FPGAs [197]) are over-budget options for IoT applications. At last, for MCU deployment CMix-NN library supports convolutional kernels with variable bit-width (8, 4 and 2) that enables the deployment of 68% accurate MobileNet on an STM32H7 microcontroller unit.

2.4 Encoding

Weight encoding (or compression) is the process to reduce the memory footprint of the ConvNet parameters. There are several techniques used to solve this task, ranging from general-purpose dictionary-based coders to lossy type-specific compression algorithms. In a generic ConNet compression pipeline the encoding stage uses to be placed at the bottom node, in fact, it needs to take advantage of the sparse and quantized tensors generated by the two previous compression steps: pruning and quantization. Each sparse tensor is composed of non-zero weights and location indices (or metadata). As the non-zero weights are hard to compress, most of the gain be found by the compression of the metadata.

ConvNet tensors can be compressed using different encoding techniques. They can be grouped in *sparse-specific* techniques, and *type-agnostic* techniques. The former are designed ad-hoc to take advantage of the sparse structures, while the latter are compression algorithms designed for a generic type of data.

2.4.1 Sparse-Specific

Each sparse-based encoding algorithms mainly focus on reducing the storage of the metadata, while non-zero values usually cannot be compressed. Based on the sparsity percentage there is a different optimal sparse representation scheme, as their efficiency is strictly correlated to the sparsity.

Some of the most popular encoding schemas are:

- **BitMap**. This is the simplest scheme as the metadata stores 1 bit for each weight, which value is 1 if the weight is non-zero and 0 vice-versa. This scheme is very simple but yet effective, especially for lower sparsity levels. Figure 2.8 shows a pictorial example of Bitmap Encoding where dense values are colored green, while sparse ones are in white. The set of dense values is *nz-values*.

- **Coordinate sparse matrix format (COO)**. A very simple sparse matrix format is COO, who stores for each non-zero value its coordinates (x, y). This sparse representation does not allow direct arithmetic operations nor slicing, but it ensures a very fast conversion to other sparse formats, like CSR/CSC which, instead, allow direct operations. The main advantages of this format are the simplicity to implement, and the facility to manage the sparse indices. On the other hand, this format is more suitable for high sparse regimes, as it does not always guarantee high compression rates. Figure 2.8 shows a pictorial example of the COO format. Here the cardinality of *nz-row* and *nz-col* indices are the same of the *nz-values* set.

- **Compressed Sparse Matrices (CSR, CSC)**. These sparse representations are designed for scientific computing, in particular for 2-dimensional sparse

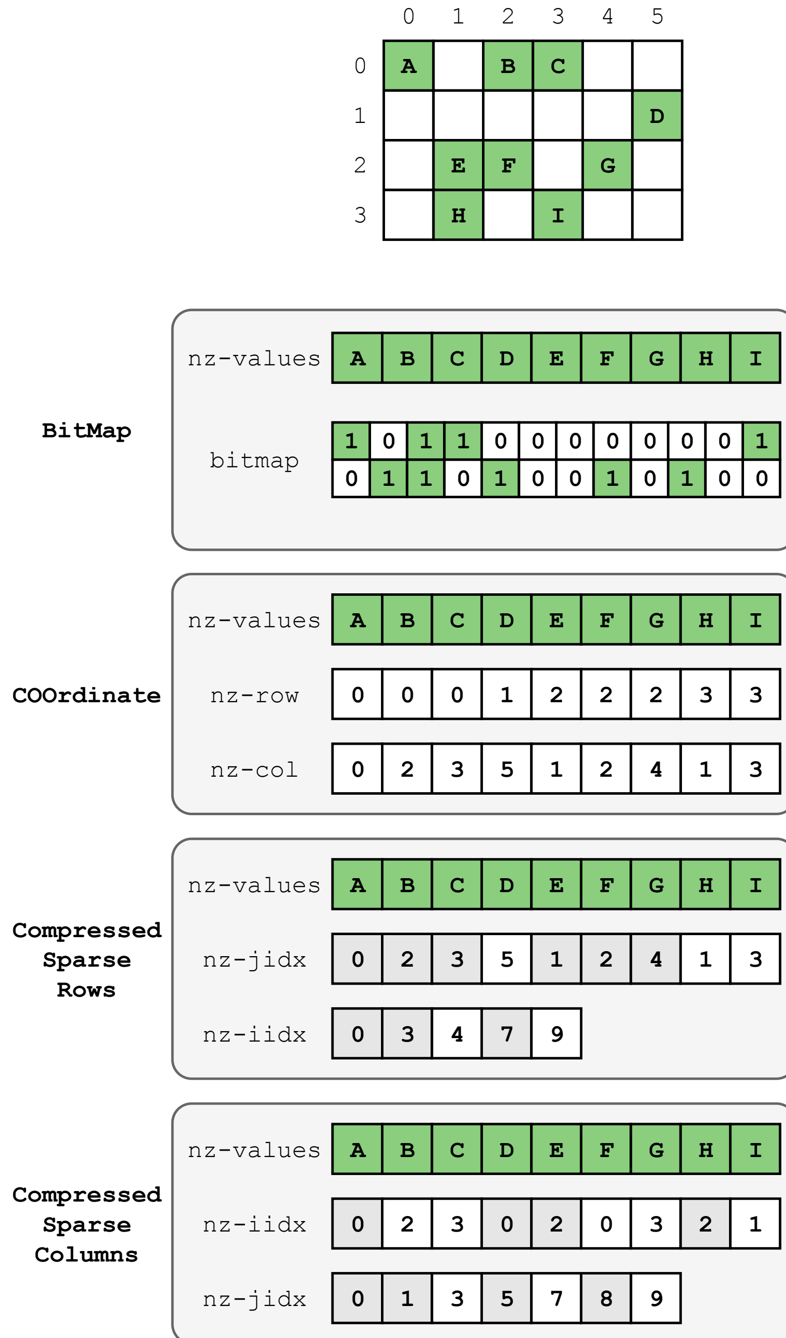


Figure 2.8: Types of sparse encoding. From top to bottom: BitMap 2.4.1, COO 2.4.1, CSR and CSC 2.4.1.

matrices. The main advantage of these sparse matrices is that they allow

being used directly in arithmetic operations: addition, subtraction, multiplication, division, and matrix power. Compressed Sparse Row (CSR) and Compressed Sparse Columns (CSC) are very similar sparse representations that store respectively the indices metadata from rows or columns point of view. Figure 2.8 shows a pictorial example of both the formats. For CSR *nz-iidx* counts the number of non-zero values for each row, while *nz-jidx* stores the column position of each non-zero value. For CSC *nz-jidx* counts the number of non-zero values for each column, while *nz-iidx* stores the row position of each non-zero value.

All the encoding types cited above have been presented considering weight level granularity to induce sparsity in the ConvNet, but they can be still extended to exploit bigger chunks of zero elements grouped in blocks or patterns. Furthermore, sparse matrices formats (COO, CSR, CSC) can be modified to store just the delta offset of the indices between two non-zero values, with the aim to boost the compression gain.

2.4.2 Type-Agnostic

- **Run-length Encoding (RLE)**. This is a form of lossless compression where sequences of equal and contiguous elements (*runs*) are stored as a single data value with the size of the sequence. This type of encoding becomes more efficient in such cases where many of these runs are present. In this particular case of study, which is sparse quantized matrices, the runs to compress are composed of zero values. Usually, this particular technique can be used as a point of beginning in the design of a more complex suitable algorithm for sparse compression and inference.

- **Huffman Coding** [**94**] is a type of optimal prefix code usually used for generic lossless data compression. The algorithms encode source symbols using variable-length codewords to generate a table. This table is derived from the estimated frequency of occurrence of each source symbol. This algorithm is also an entropy-based coder, and so it uses to represent more common symbols with fewer bits, while less common symbols with more bits. A popular example of its application on ConvNet compression is showed in [68], where the authors first prune and then quantize multiple ConvNets to maximize the compression.

- **LZ4**[**136**]. This lossless data compression algorithm belongs LZ77 family of byte-oriented compression schemes. The main difference between LZ4 and other LZ77 algorithms based on DEFLATE [33] is that LZ4 does not involve Huffman Coding in its compression pipeline, but it uses only a dictionary-matching. This technique does not aim to reach maximum compression rates,

but instead, it focuses on decompression speed: a crucial for online decompression routines. Compared to other compression algorithms, LZ4 shows a good trade-off between compression rate and decompression speed. However, the extreme decoding speed is certainly its main merit: up to multiple GB/s per core, typically reaching RAM speed limits on multi-core systems.

- **ZLIB**[34]. This is not a simple compression algorithm, but instead a full software library for general-purpose data compression. Its compression pipeline is based on DEFLATE, and it stores multiple blocks composed of bytes of source data with relative headers. It provides multiple compression configurations, trading off between compression speed and compression rate. The maximum compression configuration allows Zlib to reach extreme compression rates, at cost of relatively slow decompression. Above the huge compression capability, its main merit is the portability across platforms.

2.4.3 Inference of Encoded ConvNets

Once the ConvNet parameters are encoded in a particular format, they are stored in the memory of the edge device memory: for example the FLASH memory for general-purpose micro-controller units. The main issue is how to process the model weights, as they cannot be directly multiplied for the input samples. At this time, there are two main ways to manage the inference stage at the edge.

Direct Inference

The first approach is to direct process the data in the encoded format, which means avoiding the decompression stage. To operate this method the inference engine needs to be designed ad-hoc for the encoded format. This can be done both on (i) general-purpose MCU, with specialized sparse kernels, and on (ii) hardware accelerators, which require custom hardware components co-designed with specialized processing routines. In both cases, the main benefits to use direct processing of compressed data are two: avoiding unnecessary data transfer (fast inference) and reducing the compressed model size (tiny storage). To develop an efficient encoding algorithm to run-time process compressed data, the crucial requirements to respect are surely the compatibility with the inference engine and the compression rate. In fact, the decompression speed is not relevant for this type of approach, as they totally skip the decoding stage.

One example of sparse kernels for MCU is showed in [225], where the authors use CSR format to on-the-fly process the encoded ConvNet layers, bypassing the decompression phases. On the other hand, an example that exploits CSR-encoded ConvNets of hardware-accelerator is [70]: an efficient inference engine working with customized sparse matrix-vector multiplication and weight sharing. In [157] instead the authors designed an accelerator architecture able to fully take advantage of

the sparsity of both weights and activations. In this case, the compressed format includes, besides the standard non-zero values array, an index vector composed by the number of dense values followed by the numbers of zeros before each value. More recently the use of systolic tensor arrays (2-D pipelined arrays of tensor processing elements) to accelerate general matrix multiplication (GEMM) of sparse ConvNets inference. They apply a novel block-shaped pruning based on density, then using a bitmap encoding on 8 elements blocks. In summary, these techniques exploit the sparse ConvNets during edge inference with a full hardware-software co-design. Then, thanks to this larger vision, they provide astonishing results in terms of latency and compression, which are hardly equalized with other hardware-agnostic approaches. Other popular works to fast inference of sparse ConvNets with custom accelerator are proposed in [9, 207, 238, 130].

At last, it needs to be observed that similar approaches have been proposed to manage sparse ConvNets also using sparse kernels on GPUs [38, 48], however, this section does not overview such approaches as the focus is on the edge inference.

Run-time Decompression

The second possible approach to manage MCU inference of compressed ConvNets is to run-time decompress the ConvNet. This approach focuses to optimize the ConvNet from a storage point of view, as it does not aim to directly speed up the inference. In fact, these approaches are designed to process a dense inference: the advantage to have sparse tensors is exploited just by the reduction of the memory footprint.

One possible approach is to layer-wise decompress the layers, which were priorly stored in FLASH memory as separated blocks, avoiding the full model decoding. An example is EAST [57], where block-sparse ConvNets are compressed using LZ4 algorithm and run-time processed on Arm Cortex M4 MCU. A similar strategy has been explored in [2]. The authors extended the Viterbi-based pruning presented in [116] with a novel encoding scheme able to compress both indices and non-zero values. This technique allows them to build a highly parallel sparse-to-dense reconstruction architecture, which is the key enabling to fast reconstruct the dense model during the inference. They use a custom pruning algorithm, which selects the pruning candidates according to how much they fit the Viterbi encoding; furthermore, they need a custom architecture to process the run-time decompression during inference. Using this type of strategy, above the compression rate, makes crucial the decompression speed: in fact, this feature needs to bring negligible latency overhead to the inference stage. For this reason, the choice of the compression algorithms needs to be driven by its decompression speed, as too high latency overheads might frustrate the memory footprint benefit due to compression.

A different strategy to accomplish run-time decompression is the on-the-fly generation of ConvNets during inference. Various works explored the factorization of

convolutional filters to compressed model representations, these solutions exploit these compressed models to generate the full structure just run-time during inference. One of the pioneering works is proposed in [65], which uses an auxiliary network, called *HyperNetwork*, to generate the weights of each layer in the main network. It is a relaxed form of weight sharing across layers, as it can generate multiple networks using just an embedding of their parameters. To deterministically generate convolutional filters at run-time, the authors of [166] used a dense combination of Fourier Bessel's bases. While in [43, 194, 200] the bases are formed orthogonal variable spreading factor (OVSF) binary codes. At last, in [217] the filters are layer-wise samples from a single learnable filter. In this way the weights are shared across different filters, enabling faster and deterministic processing of the network.

Chapter 3

Statistical-Oriented Training and Compression

3.1 Motivation

The pioneering works of efficient deep learning proposed post-training procedures to optimized ConvNets. At first, we started to work on automatized compression of ConvNets with the aim to fuse training and optimization in one unique loop. The first statement was very simple: a smaller model requires less memory to be stored and fewer operations to be executed. We analyzed how the distribution of the weights of a ConvNet model can be approximated in order to reach a good compression rate, limiting the accuracy loss. This section depicts two different works focused on the same topic: statistical-oriented training and compression.

The first strategy we present is a compression-driven training framework that aims to concurrently run training and optimization together [59]. It consists of an iterative training loop with a ternary quantization: for each layer, the weights are constrained into $(-\sigma, 0, +\sigma)$ values. To learn the boundaries $[-\sigma, +\sigma]$ we designed a layer-wise approach that allows matching the characteristics of the training set. Once the ConvNets are compressed, they show a peculiar discrete distribution of weights that can be easily stored into very small memory blocks thanks to appropriate encoding routines. In particular, in this work, we applied a custom encoding scheme inspired by the popular RLE. Furthermore, the ternary quantized layers bring sensitive benefits to computation efficiency. Indeed, the accumulation of the results produced by the multiplication of input operands with the constant parameters $-\sigma$ and $+\sigma$ can be transformed into a factorized multiplication between the constant σ and the accumulation of (eventually sign-flipped) operands. The proposed compression pipeline is validated both on ConvNets and Recurrent Neural Networks (RNN), the first applied on image classification (using the CIFAR-10 and CIFAR-100 data-sets), the second on text-based sentiment analysis (IMDb

dataset). Experimental results are promising, in fact, the achieved compression ratio ranges from 35× to 95× and the total number of matrix-vector multiplications is reduced by 99%, with an acceptable accuracy degradation: a minimum of 0.68% loss for the more overparametrized ConvNets. However, since the accuracy loss is unconstrained, some narrower and more fragile models experienced large accuracy degradation, up to a maximum loss of 18.7%.

Observed these results, we worked to improve over the original training algorithm presented in [59] introducing a knob to control accuracy. The second work is a Layer-Wise Compressive Training [58]: it extends the previous compression pipeline with the rationale to apply the σ-constrained compression only on a specific subset of layers. To address this issue we design a heuristic search to quantify the *significance* of each layer, i.e., how the layer affects the classification performance. The resulting compressed ConvNet is a hybrid model where less significant layers (those that contribute less to the inference process) are optimized with the compressive algorithm proposed in [59], whereas the most significant layers remain untouched. Experimental results demonstrated that the proposed hybrid compression algorithm is very effective to outperform previous techniques achieving a better compression-accuracy trade-off.

3.2 A Compression-Driven Training Framework

It is well known by literature that pruned weights show a bimodal distribution [72], with positive and negative samples having similar centroids and shapes (Figure 2.2). For this reason, it is possible to constraint the training into a symmetric ternary space ($-\sigma$, 0, $+\sigma$), where the σ value is learned through the SGD algorithm (in this particular case we used Adadelta version). In this way, the learning procedure directly searches the optimal σ. This approach (i) avoids an exhaustive search throughout all the possible values of σ, and (ii) allows σ to evolve and adapt by following the minimization of the loss function. The crucial feature of the σ learning is that one different σ centroid is computed for each layer, computed independently on its weight distribution. A particular case of study can be applied at coarser granularity on small-size ConvNets, merging the distribution of all the layers together in a single instance, so working at a coarser granularity.

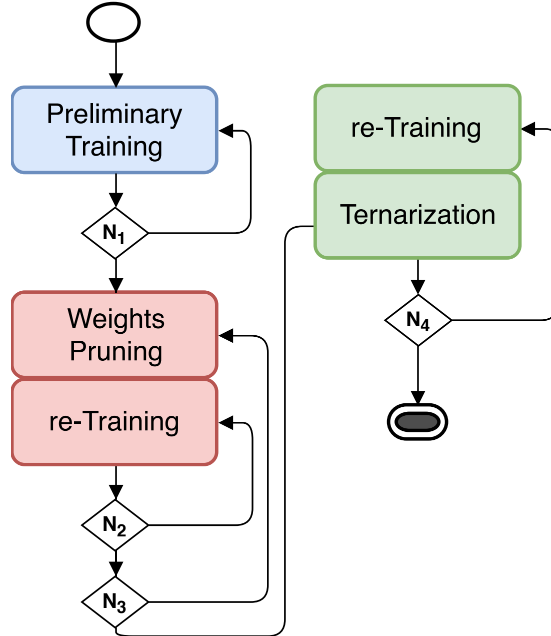


Figure 3.1: The proposed training flow.

3.2.1 Training ConvNets in a Constrained Space

Figures 3.1 summarizes the three stages of the proposed framework.

Preliminary training: a standard dense training procedure with SGD optimization to learn the weights set θ. The number of training epochs N_1 is a fixed hyperparameter depending on the task. The model can be initialized as zero, or instead from a pre-trained model by *transfer learning*. Once computed the N_1

epochs, the model weights have a distribution similar to the one shown in Figure 3.2-. This particular case refers to the second layer of AlexNet ConvNet trained on the CIFAR10 data-set. It is easy to see how the weights are shaped like a normal distribution with a zero-placed centroid.

Pruning: a magnitude-based unstructured pruning of the trained weights. We applied iterative pruning and training pipeline with polynomial decay scheduler to gradually achieve the desired amount of sparsity. This stage lasts N_3 epochs. It needs to be observed that each pruning step is followed by a short retraining phase to adapt the dense weights (remaining connections) to the new sparsity space. This dense retraining stage lasts N_2 epochs. Both N_2 and N_3 hyperparameters strongly depend on the model size and the task complexity. To enable a fine-grained exploration of the solution space, an additional parameter is leveraged to slow down the descent by multiplying derivatives by a factor $\gamma \ll 1$, which is a tunable parameter that can be included in the definition of η (Section 2.1.1). Pruning effects on weights distribution are sensitive, as shown in Figure 3.2-b for the second layer of AlexNet.

Ternarization: the remaining weights are bounded across $-\sigma$ and $+\sigma$. The process is applied layer-wise to fully take advantage of the split weights distribution after pruning (as reported in the previous section). The value of σ is computed iteratively with updates defined using the back-propagation error. At each iteration, for each layer l of the DNN: (i) σ^l is updated to its near-optimal value, (ii) all the non-pruned weights are centered on the $\pm\sigma^l$ according to their signs. It needs to be noted that the pruning masks continue to be updated also in this stage, as they are not frozen after iterative pruning. Also here, there is a short retraining phase after each ternarization step.

The maximum number of retraining epochs is defined as N_4. In case the ternarized model reaches the baseline accuracy before the pipeline stops the training iteration through early-stopping. It is worth noticing that each retraining is aware of the current σ-ternarization, namely, the derivatives are computed considering the actual σ: a crucial aspect to perform a real descent and a proper accuracy-driven σ update. This is a clear difference in respect to previous work on the subject, where derivatives are not guaranteed to achieve an optimal value [168]. At the end of this process, the distribution of the weights totally change respect before, as it becomes discrete: weights are no more defined over a continuous space, but just in two non-zero values in correspondence with the found σ^l. Figure 3.2-c shows a pictorial example of the final σ-constrained weights distribution for the second layer of AlexNet ConvNet.

Ternarization: Analytical Formulation

This paragraph provides a more formal description of the ternary quantization (ternarization) stage.

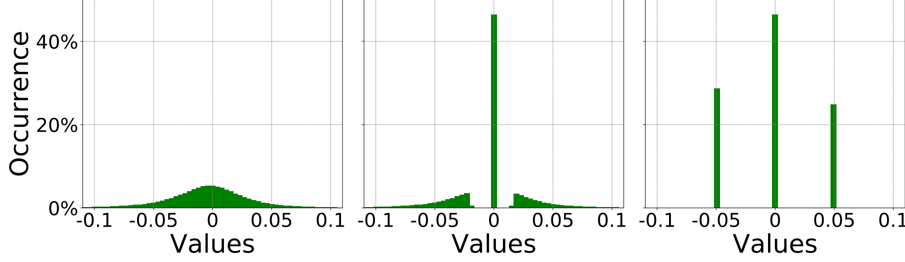


Figure 3.2: The distribution of the second layer weights of AlexNet trained on CIFAR10: after preliminary training (left), after pruning (center), after σ-ternarization (right).

Identification of σ: at any generic iteration, and independently for each layer l, the value of σ is computed from the statistical mean of the distribution of the weights θ. Such mean, for each layer, is calculated as:

$$\sigma = \frac{1}{\psi} \sum_{i=0}^{\psi-1} |\theta_i + \Delta\theta_i| \tag{3.1}$$

where ψ is the number of weights after pruning, θ_i are the weights, and $\Delta\theta_i$ is the update resulting from the back-propagation of the error function. For each iteration, the time overhead to compute σ is negligible compared to the time required to complete one training step. The positive and negative contributions of Equation 3.1 split, as shown in Equation 3.2.

$$\begin{aligned} \sigma &= \frac{1}{\psi} \left[\sum_{\theta_i > 0} (\theta_i + \Delta\theta_i) - \sum_{\theta_i < 0} (\theta_i + \Delta\theta_i) \right] \\ &= \frac{1}{\psi} \sum_{i=0}^{\psi-1} |\theta_i| + \frac{1}{\psi} \left(\sum_{\theta_i > 0} \Delta\theta_i - \sum_{\theta_i < 0} \Delta\theta_i \right) \end{aligned} \tag{3.2}$$

Following the description reported in Section 2.1.1, where $\Delta\vec{\theta} = -\eta \vec{\nabla} \mathcal{L}(\vec{\theta}_k)$, $\Delta\theta_i$ can be expressed as in Equation 3.3.

$$\Delta\theta_i = -\eta \frac{\partial \mathcal{L}}{\partial \theta_i} (\vec{\theta}_k) \tag{3.3}$$

Finally, at the $(n+1)$-th iteration, (3.2) can be written as $\sigma_{(n+1)} = \sigma_n + \Delta\sigma$, with $\Delta\sigma$ defined as below:

$$\Delta\sigma = \frac{1}{\psi} \left(\sum_{\theta_i > 0} \Delta\theta_i - \sum_{\theta_i < 0} \Delta\theta_i \right) \tag{3.4}$$

Update of σ in the constrained space: σ value is computed each iteration from the weighted arithmetic mean of the gradient components. Hence, the partial

derivative of each weight would affect the value of σ as strong as that weight is far from the optimal value. In other words, as far as using the arithmetic mean for the first step represents a reasonable starting point, to carry out an optimum search strategy during fine-tuning σ is updated in a constrained solution space. This space can be seen as a semi-bisector described as in Equation 3.5, where $\vec{s}$ is a vector having components in the form $s_j = sign(\theta_j)$ and norm $||s|| = \sqrt{\psi}$.

$$\hat{e} = \frac{\vec{s}}{||s||} \tag{3.5}$$

As the dense weights can be expressed in the vectorized form $\vec{\theta}$, all their possible values (thus including the optimal solutions) can be found along the tensor $\vec{\theta} = \sigma_n \vec{s}$. It needs to be noted that theoretically, some components of the direction could change, as θ_j might invert its sign. Therefore, the solution space is the set of all possible semi-bisectors, and the complete definition of s_j takes the form reported in Equation 3.6.

$$s_j = sign(\theta_j + \Delta\theta_j) \tag{3.6}$$

Let us assume a generic 5×5 array is pruned, keeping dense only two positive weights (θ_x, θ_y) and one negative weight θ_z are left. Then the solution space can be formulated as a 3-dimensional array and becomes a semi-straight line, whose direction is $\vec{s} = (1, 1, -1)^T$. Figure 3.3 gives a pictorial representation of the solution space taken in example. If the actual solution $\vec{\theta} = (\theta_x, \theta_y, \theta_z)^T = (\sigma_n, \sigma_n, -\sigma_n)^T$ belongs to the attractive valley of a minimum point H, then $-\vec{\nabla}\mathcal{L}(\vec{\theta})$ will point towards it.

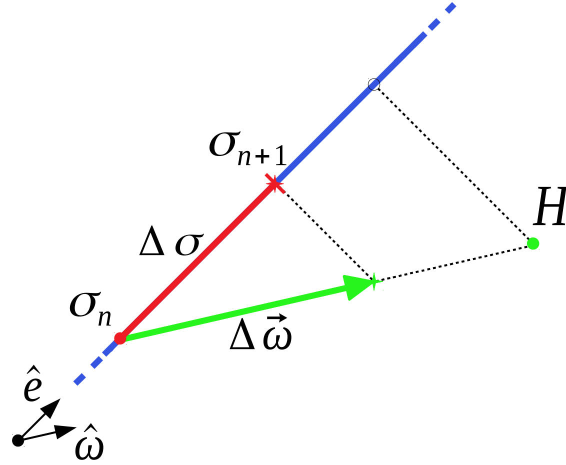


Figure 3.3: Visual representation of the solution space.

However, $\vec{\theta}$ can only move along $\vec{s}$, i.e., $\hat{e}$. This means that it is possible to find the projection of $-\eta\vec{\nabla}\mathcal{L}(\vec{\theta})$ on $\hat{e}$ to known which is the optimal $\Delta\sigma$, i.e., the closest

point to H. This is the main objective of this algorithm: adapting σ to approach the minimum point in the best *approximated* way. In fact, the scalar product in Equation 3.7 that projects $\Delta\vec{\theta} = -\eta\vec{\nabla}\mathcal{L}(\vec{\theta})$ on $\hat{e}$ can be used to update σ_n, obtaining a formulation which is similar to (3.4).

$$\Delta\sigma = \langle\vec{e}, \Delta\vec{\theta}\rangle = \frac{1}{\sqrt{\psi}}\langle\vec{s}, \Delta\vec{\theta}\rangle = \frac{1}{\sqrt{\psi}}\sum_{i=0}^{\psi-1} s_i \Delta\theta_i \tag{3.7}$$

The difference between (3.4) and (3.7) stands in a constant factor, $1/\sqrt{\psi}$, which can be a good starting point for tuning the slowing factor γ previously described.

Multiplication savings

A factorization of σ from the matrix of weights θ can significantly reduce the computational load. Indeed, σ can be pre-multiplied by the input $\vec{x}$ to compute $\vec{z} = \sigma\vec{x}$, and then to calculate the dot-products $\theta \cdot \vec{x}$ using $\vec{z}$ and S, like showed in Equation 3.8. It needs to be noted that S is the matrix whose entries belong to the set $\{-1, 0, 1\}$.

$$\vec{y} = \theta\vec{x} = (\sigma S)\vec{x} = S(\sigma\vec{x}) = S\vec{z} \tag{3.8}$$

Thank this straightforward transformation, most of the multiplications can be replaced by sums.

For the sake of comprehension, the calculation for the i-th element of the output vector $\vec{y}$ is reported in Equation 3.9, where $Z = |\vec{z}|$ is the cardinality of $\vec{z}$.

$$y_i = \sum_{j=0}^{Z-1} s_{ij} z_j = \sum_{\forall s_{ij}=1} z_j - \sum_{\forall s_{ij}=-1} z_j \tag{3.9}$$

Weight Encoding

DNNs obtained with the proposed σ-constrained training show very sparse matrices that can be efficiently compressed using some encoding scheme. The one used in this work is derived from the well-known run-length encoding.

As reported in Figure 3.4, the weights of each layer can be stored as a tensor of ternary components (−1,0,1), each of them represented as a 2-bit integer, and a common multiplicative constant σ represented as a 32-bit floating-point. The encoding algorithm parses the matrix row by row thus generating strict alternation of two basic elements: a *counter* on N bits, which replaces a sequence of zeros with its unsigned integer length, and a non-zero *weight* represented with a single bit, namely 0 for σ and 1 for $-\sigma$ (please refer to Figure 3.4-b).

While a classical run-length encoding would select N s.t. the longest sequence of zeros, i.e., $M = 2^N - 1$, can be correctly represented, our scheme accounts for the following observation. If only a few zero sequences are longer than $M = 2^N - 1$,

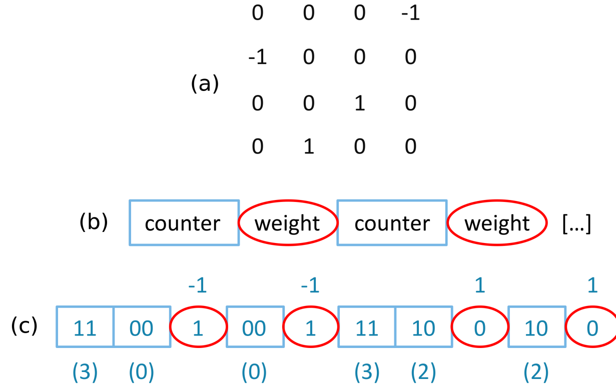


Figure 3.4: Weights matrix encoding: (a) pruned matrix, (b) encoding scheme, (c) actual memory mapping.

the use of $N+1$ bits is a huge waste of memory. Instead, it is possible to split those sequences and represent them with two, or more, counters of size N. Even if this approach breaks the strict regularity of the described pattern, this exception can still be acknowledged by the decoder by forcing the first counter to be M. To avoid errors, if the sequence of zeros is exactly of length M, a *fake* counter set to zero must be put after the one set to M. Additionally, if two non-zero weights are adjacent, a zero-counter must be included in between. However, this also represents another unlikely situation when high pruning percentages are used. As an ultimate optimization, the last counter can be omitted because the dimension of the matrix is known.

Figure 3.4-c reports an example where $N = 2$. Please notice that this encoding is independent of the precision of σ. As a final remark, two or more matrices can be concatenated, either by rows or by columns, in order to be merged together in a single compressed representation.

In conclusion, the adopted encoding allows storing the network structure parameters very efficiently, bringing most of the information in a N-bit data structure. In this regard, to quantify the compression rates achieved by the encoding algorithm, we introduced the value CR defined as the ratio between the actual memory size needed by the model after the preliminary training phase, and the storage needed after ternarization and encoding. As an example, Equation 3.10 depicts the formulation of CR using the proposed encoder with a 2-bit counter: N^0 is the number of parameters after the preliminary training (defined over 32 bits); N^σ is the number of σ constants returned after ternarization (defined on 32 bits); *length* is a function returning the bit-length of the encoded array Enc.

$$CR = \frac{N^0 \cdot 32_{bit}}{N^\sigma \cdot 32_{bit} + length(Enc)} \tag{3.10}$$

3.2.2 Experimental Results

ConvNets on CIFAR10 and CIFAR100

The CIFAR10 and CIFAR100 data-sets [109] consist of 45k images, 5k for validation and 10k for testing. They only differ by the the number of classes (i.e. labels), 10 and 100 respectively. We tested our proposed method on 5 very popular ConvNets: AlexNet [110], VGG19$_{bn}$ [180], ResNet18 and ResNet50 [77], and SqueezeNet [95].

For each ConvNet we explore 2 different pruning sparsity: 80% and 90%. Tables 3.1 and 3.2 report the obtained results. For each table, from left to right, we report the model name, the baseline (i.e. accuracy achieved with a standard dense training), the sparsity percentage, the accuracy after the compression-driven training with its difference w.r.t. the baseline, and at last the compression rate CR. It is clear to see how the residual networks ResNet18 and ResNet50 provide the best overall results: they reach up to 67× of compression with a small accuracy loss (close to 2% in the worst case), for both CIFAR10 and CIFAR100. Remarkable results can be observed also on the VGG19$_{bn}$ model: $CR = 78\times$ with 3.2% accuracy loss. On the other hand, smaller ConvNets suffer sensitively by ternary quantization, in fact, their accuracy loss is not negligible ($> 2\%$). For instance, SqueezeNet is designed to be very thin, with very small kernels. This aspect is crucial for the effectiveness of the ternarization process: a smaller kernel means lower resolution which is further reduced after the ternary quantization, resulting in a substantial loss of information. However, from a compression point of view, the ternarization is still very effective also on those thinner models: the best result, in terms of trade-off, is achieved with AlexNet on the CIFAR10 with a $CR = 67\times$ and about 6% of accuracy loss.

Focusing on the compression results of CIFAR10 and CIFAR100, it can be observed how the compression rate, at the fixed sparsity level, is virtually the same for both datasets. In fact, the ConvNet architectures used for CIFAR10 and CIFAR100 are very similar: they basically differ just from the size of the last dense layer, which manages a different number of classes. This demonstrates that the proposed compressive training algorithm can achieve very high compression rates regardless of the target application. Concerning hardware improvements, for all considered networks the average multiplication savings is above 98.5%. (Section 3.2.1). This result clearly proves that the proposed algorithm is not only able to reduce the model size, but also dramatically reduces the number of inference operations.

RNN on IMDb

We also tested our algorithm on a recurrent neural network with the aim to prove its generalization capability. In particular we used a Long Short-Term Memory

Model	**Baseline** (%)	**Sparsity** (%)	**Tern. Acc.** (%)	**CR** (×)
AlexNet	77.22	80 90	71.02 (-6.20) 69.95 (-7.27)	38 67
VGG19_bn	93.34	80 90	90.12 (-3.22) 90.05 (-3.29)	39 78
ReNet19	93.02	80 90	92.34 (-0.68) 91.96 (-1.06)	36 67
ReNet50	93.65	80 90	91.88 (-1.77) 91.42 (-2.23)	36 66
SqueezeNet	90.00	80 90	73.54 (-16.48) 72.14 (-17.86)	35 69

Table 3.1: Results on CIFAR10.

Model	**Baseline** (%)	**Sparsity** (%)	**Tern. Acc.** (%)	**CR** (×)
AlexNet	43.87	80 90	35.69 (-8.18) 35.21 (-8.66)	38 63
VGG19_bn	71.95	80 90	66.38 (-5.57) 64.12 (-7.83)	38 77
ReNet19	70.93	80 90	70.54 (-0.39) 70.07 (-0.86)	36 67
ReNet50	71.05	80 90	70.23 (-0.82) 70.19 (-0.86)	36 65
SqueezeNet	56.29	80 90	39.56 (-16.73) 37.59 (-18.70)	37 74

Table 3.2: Results on CIFAR100.

(LSTM) cell [114] on the *IMDb data-set* [137] benchmark, which is composed of 25 thousand textual reviews of movies with a positive (1) or negative label(0). An abstract view of the RNN model is depicted in Figure 3.5. The cardinality of the dataset is of 50k samples, which are equally split into $50 - 50$ (25k for training and 25k for testing). The reviews are preprocessed and each one is encoded as in a vector VEC, which is then processed by the LSTM cell; the output of the LSTM is averaged through a pooling layer and then fed to the logistic regression

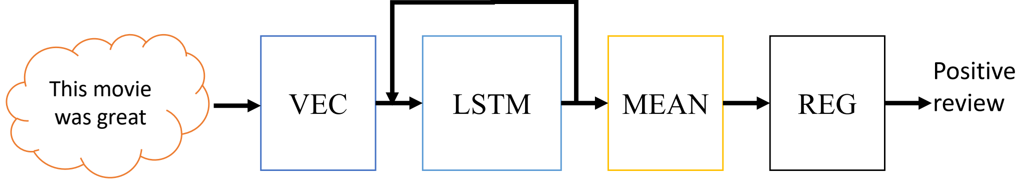


Figure 3.5: Pictorial representation of LSTM network structure used on IMDb dataset.

Baseline (%)	**Sparsity** (%)	**Tern. Acc.** (%)	**CR** (×)
88.59	50	88.67 (+0.08)	35
	80	88.36 (-0.23)	55
	90	87.21 (-1.38)	95

Table 3.3: Results for custom RNN trained on IMDb.

layer. We applied our compressive training only on LSTM layer, for sparsity$\in$ $\{50\%, 80\%, 90\%\}$.

Experimental results look promising, as shown in Table 3.3. For sparsity lower than 90% the accuracy loss is negligible ($\leq 0.23\%$), with a compression rate that ranges from 35× to 55× respectively for 50% and 90% of sparsity. For the highest sparsity level (90%) instead, the compression rate reaches 95×, at cost of higher accuracy loss (1.38%). It needs to be highlighted that with a pruning percentage of 80%, our technique is not just able to avoid any miss-classification, but it marginally improves the dense model accuracy baseline by 0.08%.

Concerning multiplication savings, we observed that the LSTM model can be grouped into matrices, each of them factorized by a dedicated σ value, which for this particular case are σ_1 and σ_2. Since both inputs and outputs have n components and matrices are squared, then just $\phi_T = 2n$ multiplications are needed instead of the originally required $\phi_O = 8n^2$ multiplications. Therefore, considering that for the specific network used in this work n is equal to 128, the total amount of saved multiplications is the ratio $\phi_T/\phi_O \approx 0.2\%$, meaning that 99.8% of multiplications are eliminated.

3.2.3 Conclusions

We proposed a novel compressive training algorithm for deep neural networks able to fuse together pruning, ternary quantization, and encoding in a homogenous optimization loop. The experimental analysis clearly demonstrated that our training algorithm is able to dramatically reduce the storage needed to store the full-precision 32-bit network ranging from 33× to 95× of compression rate. In

most cases the accuracy drops are negligible. To exploit the sparse and quantized networks, we proposed a custom sparse encoding scheme able to sensitively reduce both the memory footprint and the total amount of matrix multiplications by 99%, thus enabling an efficient deployment of deep neural networks at the edge of the IoT.

3.3 Boosting Compression via Layer-Wise Strategy

This work improves over the original algorithm described in the previous section 3.2, introducing a knob to control accuracy. It consists of a two-stage flow: first, layers are sorted by means of heuristic rules according to their significance; second, a modified stochastic gradient descent optimization is applied on less significant layers such that their representation is collapsed into a constrained subspace.

3.3.1 A Greedy Approach for Compressive Training

This section gives a step-by-step description of a new greedy strategy

This section illustrates step-by-step how the proposed greedy strategy compresses the layers during training. The flow is reported in Figure 3.6. At a glance, the algorithm is composed of three main stages denoted with different colors: *pre-training* (light red), *setup* (yellow), and *optimization* (blue). The numbered boxes serve as an index for the detailed description.

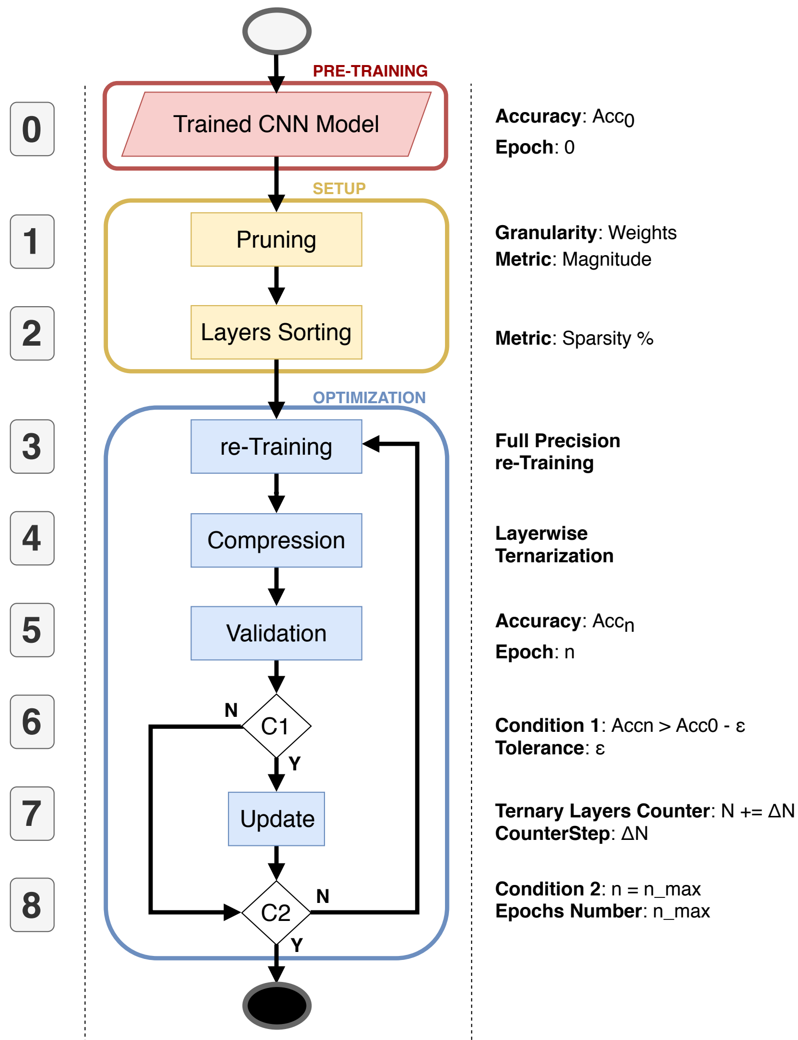


Figure 3.6: The proposed net compression pipeline.

Pre-Training

Step 0—Trained ConvNet Model. The input of the proposed flow is represented by the trained model of the ConvNet that needs to be optimized. Our solution is designed to work on classical floating-point ConvNet models; however, it can be also applied to quantized ConvNets. It can work equally on top of pre-trained floating-point ConvNet models, or on *clean models*, after a standard training process.

Setup

Step 1—Pruning. It consists of a standard magnitude-based pruning applied to both convolutional and fully connected layers. In detail, we applied pruning

with layer-wise homogeneity (see Section 2.2.4): the weights of all the layers are included in a unique set, in order to be pruned all together, computing a unique importance score, with no distinction between which layer they belong. In this way, the user specifies an a priori value for the desired percentage of the sparsity of the net, and since such a value is unique for the entire ConvNet, each layer may show a different pruning percentage. This allows representing the ConvNet model with a non-homogeneous inter-layer sparsity. We decided to follow this direction under the assumption that each layer influences the knowledge of the ConvNet differently, i.e., each layer provides a specific contribution to the final prediction. For this reason, the layers do not all keep the same amount of information, but the knowledge is spread heterogeneously among the layers, and hence they keep different percentages of redundant parameters.

Step 2—Layers Sorting. It is known that some layers are more significant than others. That means the compression of less significant layers will marginally degrade the overall performance classification. The most significant layers, instead, should be preserved in their original form. As a rule of thumb, we selected the *intra-layer sparsity* as a measure of significance. More in detail, we argue that layers with lower intra-layer sparsity are those that play a major role in the classification process, whereas those with a higher intra-layer sparsity can be sacrificed to achieve a more compact ConvNet representation. In other words, we base our concept of significance on the number of activated neurons.

A significance-based sorted list of layers is generated at first. All layers are processed as they appear in the original model, and then pruned and sorted based on their weights distribution according to the rule *higher-sparsity first-compressed*. A graphical example is reported in Figure 3.7, where (*i*) the top-most pictures represent the original weight distribution of each layer (numbered L1 to L8) of the AlexNet model trained on the CIFAR10 dataset; (*ii*) the plots in the middle depicts the weight distribution after pruning; and (*iii*) the down-most plots report the layers sorted according to their significance, namely, less important layers are those with a smaller standard deviation, which is directly correlated to their sparsity.

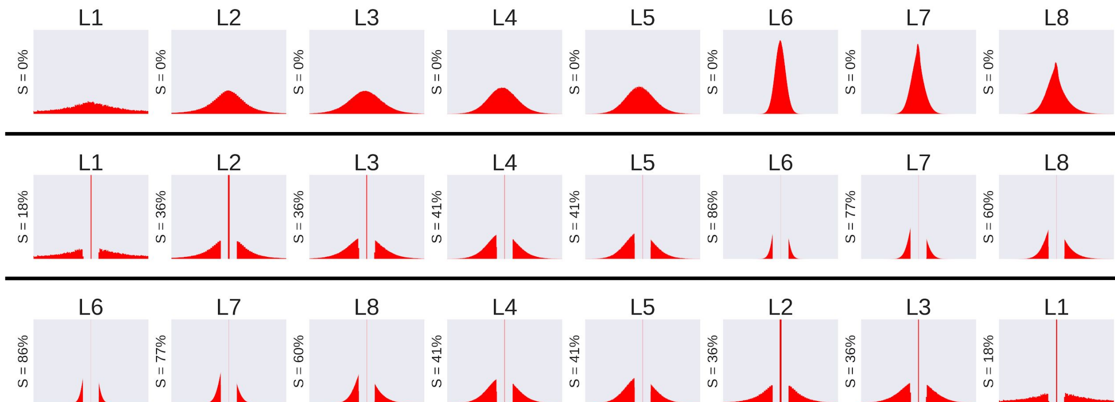


Figure 3.7: AlexNet on Imagenet, layers after the sorting algorithm; the sparsity value S is reported for each layer.

Optimization

Step 3—re-Training The retraining phase is applied in order to recover the accuracy loss due to pruning. The retraining is applied after pruning at first, and then after each optimization loop.

Step 4—Compression It is the compressive training described in Section 3.2. The weights are projected in a sub-dimensional space composed by just three values for each layer, i.e., $(-\sigma, 0, +\sigma)$, with σ defined layer-wise.

Step 5— Validation The model is validated in order to quantify the accuracy loss due to compression, and thus to decide if it is worth continuing with further compressions. Validation is a paramount step for the greedy approach as it actually enables an accuracy-driven optimization. The accuracy Acc_n is evaluated and stored after each compression epoch n.

Step 6—Condition 1 (C1) The accuracy recorded during the n-th epoch (parameter Acc_n) is used to determine if the ConvNet model can be further compressed, as in Equation 3.11. The accuracy of the pre-trained model (Acc_0) works as a baseline, whereas the parameter ϵ represents the user-defined accuracy loss tolerance:

$$Acc_n > Acc_0 - \epsilon. \tag{3.11}$$

It is worth noticing that the higher the ϵ, the higher the compression of the ConvNet model[1]. The framework takes a larger execution time for small values of ϵ; this is due to the increased complexity in selecting a good combination of layers

[1]The optimal value of epsilon can be found through several hyperparameter optimization techniques. However, in this work, we fixed the ϵ value at 1%, considering significant solutions just the ones with negligible accuracy loss.

that allows matching the user's constraints. However, both the execution time and the power consumption of the inference stage are totally uncorrelated to the ϵ value.

$C1$ can lead to two possible branches: if Equation 3.11 holds true, then the algorithm goes to step 7; otherwise, the quit condition $C2$ is evaluated.

Step 7—Update This stage is applied *if* Equation 3.11 is verified. The counter N indicates how many layers of the sorted list can be compressed. Each and every time $C1$ is evaluated as true, and N is incremented by ΔN. The latter represents another granularity knob, hence on the speed of the framework; ΔN is mainly defined by the network size: the larger the ConvNet model, the larger the ΔN.

Step 8—Condition 2 (C2) This last condition is based on the maximum number of epochs n_{max}, a user-defined hyperparameter. At the n-th iteration, if more than n_{max} epochs are elapsed, the algorithm stops, else the flow iterates over step 3.

Figure 3.8 shows the accuracy evolution during the optimization loops, reporting the validation accuracy both after an entire training epoch (blue line) and inside the optimization loop of the network (red line), that is retraining and compression. To better understand the behavior of the model during its compression, we recall that, for each loop, the algorithm first runs a full precision dense training step that updates the weights to recover the accuracy (Step 3), and then it compresses the weights into the σ-constrained solution space (Step 4). This last step is the main cause of the accuracy drop (i.e., the gap between red and blue lines). As the plot suggests, the accuracy loss is recovered within each retraining phase. The peak of loss reflects the addition of a new layer in the compressible subset list. There are layers that influence more the performance drop, but in general, after some epochs, the network reconstructs the information lost.

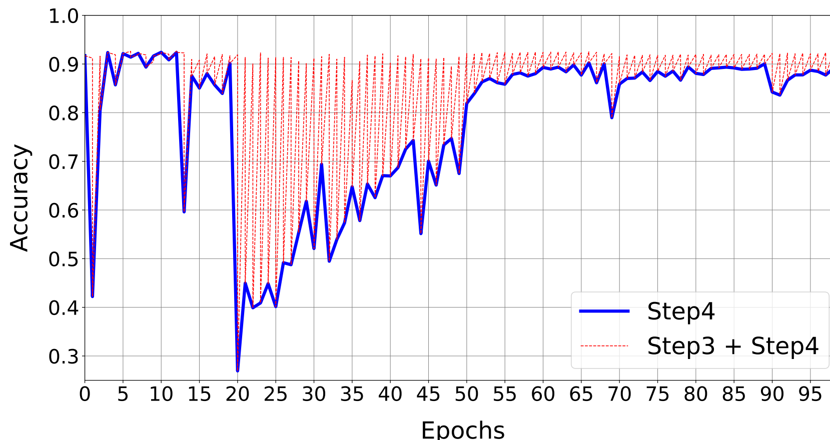


Figure 3.8: Accuracy evolution during the training of VGG-19 architecture on CIFAR10 dataset. The plot shows all the details of the optimization loop. The blue line is the accuracy after each training epoch (Step4), which is the accuracy of the compressed model (one value for each training epoch). The red dotted line depicts the accuracy evolution inside a full training step (Step3 + Step4) to highlight how the model is able to rapidly recover the accuracy loss after each compression step, using just one epoch of retraining.

3.3.2 Experimental Results

The objective of this section is to quantify the effectiveness of our compression training w.r.t. other state-of-the-art solutions. We focus on some of the most popular ConvNets models and datasets.

ConvNet models—The adopted ConvNet models are trained from scratch or retrained from the *Torchvision* package of PyTorch [158]. More specifically, we adopted the following ConvNets: AlexNet [110], VGG [180], and several residual networks with increasing complexity [77].

All the models are trained and tested using PyTorch [158] (version 0.4.1). The training epochs of the compressive algorithm are fixed to 100, with a batch size of 128 and an initial learning rate of 1×10^{-3}, which is scaled every 33 epochs by a factor of 0.1.

Datasets—We used three different datasets for the experimental analysis: CIFAR10, CIFAR100 [109], and Imagenet [110]. As the datasets are composed of images with different sizes, each architecture has been adapted to the input dataset, in particular to the size of its samples.

CIFAR10 and CIFAR100 are two large-scale image recognition benchmarks that, as their names suggest, differ for the number of available labels. Both are made up of 60k RGB images. The raw 32×32 data are pre-processed using a standard contrast normalization. We applied a standard data augmentation composed of a 4-pixel zero-padding with a random horizontal flip. Each dataset is equally split

into 50k and 10k images for training and validation, respectively. The intersection between the training set and the validation set is void. Tested ConvNets are: AlexNet [110], VGG [180], and residual networks [77].

ImageNet (ILSVRC-2012) represents a ultra-large image dataset. Being composed of about 1.2 M images for training and over 50k ones for validation, it accounts for a total of 1k different classes. We followed the original data augmentation reported in [110]: the original raw images with size 256×256 are cropped into 224×224 patches with a global contrast normalization. For the training stage, the transformation is applied randomly together with a horizontal flip; during validation, a center crop manipulation is applied. AlexNet [110] and ResNet18 [77] are the two tested ConvNets.

Performance

For the validation of the proposed technique, we consider the trade-off between the accuracy loss and the compression ratio. The two performance metrics are described in Equations 3.12 and 3.13. The former represents the accuracy loss and is defined as the difference in terms of accuracy percentage between the original full-precision model ($Model_{FP}$) and the compressed one ($Model_{C}$). The latter describes the compression rate (CR) defined as the ratio between the memory storage needed to save the original model parameters and the storage needed for saving the compressed model after the encoding. Various evaluation metrics are used to quantify the performance of a classifier; the choice of the correct one depends on the context and by the application. However, regarding Image Classification, the most used metric is certainly the classification accuracy, which is the number of the corrected classified input samples divided by the total number of samples. For this reason, the trade-off between the accuracy loss (minor is better) and the compression rate (higher is better) can be used as the measure of the quality of the model compression, as largely employed in the literature [67]. On each compressed layer, we apply the weights encoding illustrated in [59], using a 4-bit counter. In the original model, all the weights (N_0) have to be saved in 32-bit; for the compressed model, only the different N^{σ} values (one per each compressed layer) and the total number of weights of the full-precision layers N^{FP} need 32-bit precision.

$$AccuracyLoss \quad [\%] \quad = \quad Accuracy(Model_{FP}) - Accuracy(Model_{C}), \tag{3.12}$$

$$CR \quad [\times] \quad = \quad \frac{N^{0} \cdot 32_{bit}}{N^{FP} \cdot 32_{bit} + Length(Enc) \cdot 1_{bit} + N^{\sigma} \cdot 32_{bit}}. \tag{3.13}$$

We first focus on the performance on CIFAR10 and CIFAR100 datasets with $AlexNet$, $VGG19_{bn}$, and $ResNet110$. Tables 3.4, 3.5 summarize the obtained results. In both tables, the first row reports the name of the ConvNet model with the baseline accuracy in parentheses; the second row reports the experimental results in

Model	**Baseline** (%)	**Accuracy** (%)	**CR** (×)
AlexNet	77.22	76.44 (-0.78)	26.4 ×
VGG19_bn	93.02	92.20 (-0.82)	6.7 ×
ResNet110	93.81	93.32 (-0.49)	8.0 ×

Table 3.4: Experimental results on CIFAR10. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR.

Model	**Baseline** (%)	**Accuracy** (%)	**CR** (×)
AlexNet	44.01	43.47 (-0.54)	26.4 ×
VGG19_bn	71.95	71.62 (-0.33)	6.5 ×
ResNet110	71.14	70.80 (-0.34)	7.3 ×

Table 3.5: Experimental results on CIFAR100. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR.

terms of Top-1 accuracy and accuracy loss; the last row describes the compression rate after weights encoding for each different ConvNet model. Obtained numbers refer to a user-defined accuracy loss of $< 1\%$. The numbers suggest not only that the accuracy constraint is successfully met, but they also indicate that a very large compression rates (e.g., 26.4× for the AlexNet model) are easily achieved. This proves the adopted rationale is sound and also applicable to very complex ConvNet model; it allows to preserve useful information just removing redundant, or less significant, parameters on the less significant layers.

To further understand our approach, we detail two network structures before and after the greedy compressive training. Table 3.6 shows the ResNet20 architecture on CIFAR10: the two first columns show the layers ID and their input size; the next columns show the intra-layer sparsity percentage and the bit-width adopted to represent the weights, both for the full precision model (FP) and the compressed model (CM). The last rows report the sparsity of the resulting net, the compression rate w.r.t. the original model, and the final accuracy. Table 3.7 shows the same kind of metrics for AlexNet trained on CIFAR100.

Since our framework applies automatic decisions on the number of layers to compress, for each ConvNet and each dataset adopted, the results may change substantially, also depending on the accuracy constraint and the number of iterations run.

Layer	Input Shape	FP Sparsity [%]	FP Width [Bit]	CM Sparsity [%]	CM Width [Bit]
Conv1	(16,3,3,3)	0	32	41	32
Conv2	(16,16,3,3)	0	32	18	2
Conv3	(16,16,3,3)	0	32	45	32
Conv4	(16,16,3,3)	0	32	23	2
Conv5	(16,16,3,3)	0	32	24	2
Conv6	(16,16,3,3)	0	32	31	2
Conv7	(16,16,3,3)	0	32	37	2
Conv8	(32,16,3,3)	0	32	32	2
Conv9	(32,32,3,3)	0	32	45	2
Conv10	(32,16,1,1)	0	32	29	32
Conv11	(32,32,3,3)	0	32	44	2
Conv12	(32,32,3,3)	0	32	50	2
Conv13	(32,32,3,3)	0	32	53	2
Conv14	(32,32,3,3)	0	32	62	2
Conv15	(64,32,3,3)	0	32	44	2
Conv16	(64,64,3,3)	0	32	58	2
Conv17	(64,32,1,1)	0	32	50	32
Conv18	(64,64,3,3)	0	32	70	2
Conv19	(64,64,3,3)	0	32	87	2
Conv20	(64,64,3,3)	0	32	90	2
Conv21	(64,64,3,3)	0	32	94	2
Fc1	(10,64)	0	32	39	32
Final Sparsity		0.00%		68.16%	
Compression Rate		- ×		6.1×	
Accuracy		93.02%		92.47%	

Table 3.6: Analysis of the sparsity and bit-width variation across the layers, before and after the compressive greedy training. On the left, the full precision model (FP), on the right the compressed model (CM), both referring to ResNet20 ConvNet trained on CIFAR10 dataset. The input shapes of the layers are (n, c_{in}, k_h, k_w), and (n, c_{in}) respectively for convolutional (Conv) and fully-connected layers (Fc). The height and the width of the kernels are defined as k_h and k_w, the number of input channels as c_{in}, and the batch-size as n.

Comparison with the State-of-the-Art

The analysis includes some of the most popular works on aggressive ConvNet compression: *Xnor-Net* [168], by Rastegari et al., where both filters and feature

Layer	Shape	FP		CM	
		Sparsity [%]	**Width [Bit]**	**Sparsity [%]**	**Width [Bit]**
Conv1	(64,3,11,11)	0	32	39	32
Conv2	(192,64,5,5)	0	32	57	2
Conv3	(384,192,3,3)	0	32	68	2
Conv4	(256,384,3,3)	0	32	55	2
Conv5	(256,256,3,3)	0	32	66	2
Fc1	(10,256)	0	32	18	32
Final Sparsity		0.00%		60.61%	
Compression Rate		-×		26.7×	
Accuracy		44.01%		43.47%	

Table 3.7: Analysis of the sparsity and bit-width variation across the layers, before and after the compressive greedy training. On the left, the full precision model (FP), on the right the compressed model (CM), both referring to AlexNet ConvNet trained on CIFAR100 dataset. The input shapes of the layers are (n, c_{in}, k_h, k_w), and (n, c_{in}) respectively for convolutional (Conv) and fully-connected layers (Fc). The height and the width of the kernels are defined as k_h and k_w, the number of input channels as c_{in}, and the batch-size as n.

maps are compressed in a binary space; *Ternary Weights Network* (TWN) [41] where Li and Liu et al. overcame the binary solution space adding the zero value as a third quantized instance; *Trained Ternary Quantization* (TTQ) [237], where Zhu et al. propose a new ternary quantization procedure able to use just 2-bit weights (with relative scaling factors) during ConvNet inference; *DoReFa-Net* [236], where Zhou el al. explored hybrid ConvNets with different quantization widths for weights, gradients, and activations type with binary weights and 32-bit activations; we focus on the *1-32-32 DoReFa-Net* in particular. For all the comparisons, the baseline is the accuracy obtained with full-precision models found in the PyTorch repository. In the following text, we use the accuracy loss as the main metric for comparison. Indeed, the results reported in the previous works do implement any encoding scheme, and comparing the compression rates might result in being quite unfair.

Let us first consider the results of the CIFAR10 dataset. The first row in Table 3.8 describes the ResNet20 and ResNet56 baseline accuracies; each column reports the obtained results. For the first ConvNet model, our technique is able to outperform TTN's solution with just 0.55% of accuracy loss, whereas, for the ResNet56, the solution is closer to the baseline. However, for both networks, we set up the accuracy tolerance ϵ at 1%, reaching a considerable compression rate (6.1×

Model	Baseline	Technique	Accuracy (%)	CR (×)
ResNet20	93.02%	**OUR**	**92.47 (-0.55)**	6.1
		TTN [237]	91.13 (-1.89)	-
ResNet56	93.65%	OUR	93.04 (-0.61)	6.9
		TTN [237]	**93.56 (-0.09)**	-

Table 3.8: State-of-the-art comparison of our technique with TTN work [237] with CIFAR10 dataset. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR, and it is reported just for our solution. For each comparison, we report in bold the solution with higher accuracy.

Model	Baseline	Technique	Accuracy (%)	CR (×)
Alexnet	56.55%	OUR	55.20 (-0.55)	10.9
		TTN [237]	**57.50 (+0.95)**	-
		TWN [41]	-	-
		XNOR-Net [168]	44.20 (-12.35)	-
		DoReFa [236]	53.90 (-2.65)	-
ResNet56	69.76%	**OUR**	**67.90 (-1.86)**	3.6
		TTN [237]	66.60 (-3.16)	-
		TWN [41]	61.80 (-7.96)	-
		XNOR-Net [168]	51.20 (-18.56)	-
		DoReFa [236]	-	-

Table 3.9: State-of-the-art comparison of our technique with TTN work [237] with Imagenet dataset. For each ConvNet model, the accuracy loss is referred to the baseline accuracy and it is reported in parentheses. The compression rate is abbreviated to CR, and it is reported just for our solution. For each comparison, we report in bold the solution with higher accuracy.

and 6.9×).

Table 3.9 reports the experimental results obtained with the Imagenet dataset. We can see that, for AlexNet, the best solution is that achieved with TTN. In fact, they claim to be able to improve over the full precision model. Their solution consists of a ternary weights ConvNet model (with relative scaling factors), except for the first and last layers, which are kept on float32 precision. Our solution outperforms all the other solutions, e.g., a 12.35% delta w.r.t. XNOR-Net. On the other hand, with the ResNet18 architecture, our model is able to outperform all considered state-of-the-art techniques, reaching just 1.86% of accuracy loss. For this set of experiments, considering the 1k-labels dataset complexity, we fixed the

accuracy tolerance ϵ at 2%, reaching compression rates of 10.9× for AlexNet and 3.6× for ResNet18.

Regarding the computational effort, all these techniques have negligible overheads compared to a standard training step. In fact, involving a compression stage based on extreme quantization brings negligible rises of the computation effort and time, which instead are strongly dominated by the backpropagation algorithm.

3.3.3 Conclusions

In this work we explored a layer-wise compressive training able to significantly boost the compression on ConvNets, still guaranteeing minimal accuracy losses (below 1%). The main contribution is to leverage a heuristic search to select and compress the most appropriate layers, in terms of accuracy and compression benefits. Experimental analysis shows promising results, as our technique overcomes the limitation of more agnostic pruning approaches, enabling a smarter strategy to drive the ConvNet compression. Despite the remarkable achievements shown in the paper also compare our technique with other state-of-art solutions, there is much room for improvement. First, the pruning algorithm used to induce sparsity in the selected layers can be extended with more complex patterns and/or with additional parameters with the aim to furtherly increase the accuracy vs. compression trade-off. Second, the setup of hyperparameters can be optimized, as there may exist an optimal setting that is strictly dependent on both the dataset and the architecture. Finally, the use of weight quantization represents an orthogonal knob to the proposed technique, able to brings additional savings in terms of both memory storage and computational resources.

Chapter 4

Hardware-Driven Training and Compression

4.1 Motivation

The migration from cloud services to low-cost IoT applications is challenging as it requires a full comprehension of the hardware characteristics and requirements. To effectively tackle this task the compression pipeline needs to be hardware-aware, a feature not entirely accomplished by the previously cited statistical-oriented strategies. As highlighted in Chapter 2, the main limitations of low-power IoT devices are the lack of capacious memory and storage, which have to be the focus of the design of a hardware-oriented compression and training pipeline. For example, considering the Cortex-M MCUs by ARM (Figure 4.1), the available RAM is up to 512kB, a severe bottleneck to process state-of-the-art ConvNets which usually require 10^4Mops 2. On the other hand, the on-chip storage does not exceed 2MB, and off-chip memory supports are usually not integrated and when available they affect negatively several processing metrics (latency, energy, integration cost, endurance, reliability). Considering these two gaps, the ConvNets need to be compressed fusing the optimization for both sides. This requires a more conscious pipeline able to address the hardware requirements still preserving the accuracy. The main difference between this kind of compression strategy respect aforementioned is that here the compression faces real deployment problems, which may not appear in theoretical compression strategies. The tools used are still the same as before, ranging from pruning, quantization, and weight encoding, but here the hardware device becomes the center of the compression pipeline. In this chapter, we present three different strategies to train and compress a ConvNet from a hardware perspective. The first work analyzes and assesses the memory-accuracy solution space fusing filter pruning and quantization, the second introduces a novel technique to compress ConvNets under memory (RAM) constraint, while the third proposes a novel strategy to compress ConvNets under storage (FLASH) constraint.

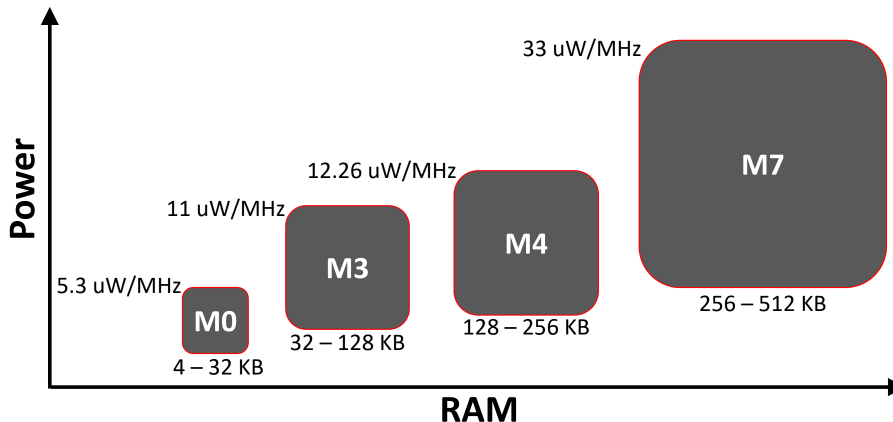


Figure 4.1: Cortex-M family: Active power and on-chip RAM size.

Among all the possible techniques to compress a ConvNet filter pruning and weight quantization have become standard strategies to reduce both memory and storage requirements. They are usually used in combination to store a lighter model with fewer operations. While filter pruning directly reduces the cardinality of the tensors of each layer, weight quantization reduces the numerical precision of the parameters. It needs to be observed that these techniques are usually *accuracy-driven*, in fact, they should provide solutions able to guarantee the highest memory compression with the lowest accuracy loss, ideally zero. However, how to fuse these two techniques is not trivial: there may be an optimal solution between the number of parameters to prune and the bandwidth to use for scaling their precision. However, there is a lack of a closed-form solution able to describe the dynamics of the learning flow, which makes the optimization loop with pruning and quantization uncertain and slow. Furthermore, an additional level of complexity is brought by the hardware constraints, which rise the complexity of the problem. This aspect becomes crucial in real-life scenarios applications, like the ones deployed on RISC cores. Since the optimization function is unknown, the selection of the best solution able to respect the user-defined constraint is blind. Intuitively, there may be two optimized models with the same accuracy but different memory footprints, and vice versa there may be two models with the same compressed memory but sensitively different accuracy levels. Hence, an optimization loop that does not consider these aspects might result in poor in finding the best solution able to meet a defined constraint, like return a compressed model with no accuracy loss but still not fitting the target memory. Finding the right balance between pruning and quantization is still an open issue, as they both affect the same source of information.

This may suggest that design space exploration would be more reliable than multi-objective optimization, but an exhaustive search is impractical due to the huge number of hyper-parameters to tune. This calls for the use of smart heuristics. At last, it needs to be remembered how quantization below 8-bit remains a

theoretical approach, as not suitable for most of the off-the-shelf low-power devices, as highlighted in Chapter 1. In view of the above considerations, it is clear how the accuracy-driven unconstrained compression presents several gaps to meet the needs of IoT real-cases scenarios. To overcome this issue we studied the distance between theoretical and practical ConvNet implementations [56] (Section 4.2). In particular, we assess the optimality of the optimization of compressed ConvNet to deploy on MCUs under memory constraints. To perform this analysis we design a novel two-stage pipeline composed of pruning and quantization: *Prune-and-Quantize* (**PaQ**). The key feature of the optimization pipeline is a novel memory-driven heuristics able to explore the memory-accuracy solution space efficiently. The framework is tested on three real-case scenarios tasks for IoT domain: Image Classification (IC) on CIFAR10 [109], Keyword Spotting (KWS) on Google Speech Command Dataset [208] and Facial Expression Recognition (FER) [16]. For the hardware validation, we used two commercial MCUs powered by Cortex-M cores: NUCLEO-F412ZG (M4-256kB), NUCLEO-F767ZI (M7-512kB).

As previously discussed in Chapter 1, n-ary quantization is largely applied on over-parametrized ConvNets [236, 168, 237, 41]. These extreme quantization techniques allow achieving a compression proportional to the reduction of the bit-width used to quantize the model, usually with negligible accuracy loss [165]. However, as largely highlighted in Chapter 1, the generic n-ary quantization schemes do not consider the actual hardware specification, which instead is crucial for real-life applications. This gap strongly motivates our work [160] (Section 4.3), which proposes a novel *memory-bounded* compression pipeline called *Virtual Quantization* **VQ**. Given a target device, with fixed memory (RAM) and instruction-set (8- or 16- bit for Cortex-M), **VQ** compresses a ConvNet with the same RAM footprint of the theoretic n-ary quantized model **Q**, where n is the maximum bit-width to meet the RAM constraint. In fact, to meet the memory constraint, **VQ** enables to compress and deploy a n-ary quantized ConvNet on general-purpose IoT MCUs, emulating the availability of n-ary instruction-set. The technique allows reaching the target using the bit-width supported by the instruction set by pruning some filters from the original model. Both **VQ** and **Q** allow to reach the memory target, but only **VQ** can be executed on-chip efficiently. Furthermore **VQ** allows reaching higher accuracy values compared to **VQ**, which usually needs to reduce drastically the bit-width n in order to meet the target. The experimental analysis has been conducted on the same three IoT tasks of previous work: IC, KWS, FER. The board used for the validation belongs to the Cortex-M family by ARM.

Another possible direction of optimization is to compress ConvNets under limited storage constraints. Given a fixed FLASH memory as input, the objective is to compress a ConvNet meeting the storage target possibly with no accuracy loss. The lack of storage space in low-cost microcontroller units is a common issue, in fact as seen in Chapter 1 even the smallest size ConvNets require more than a few MBs of parameters, while general-purpose MCUs usually provide less than 2MB

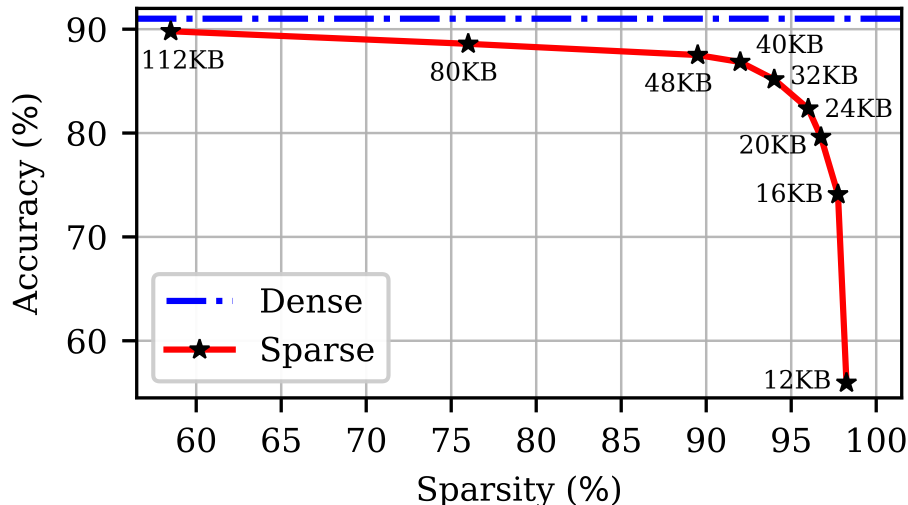


Figure 4.2: Sparsity vs. Accuracy of a compressed 9-layer ResNet under different memory constraints (the labeled numbers). The net is trained on CIFAR-10, then compressed via weight pruning and encoding. The blue dash-dotted line marks the accuracy of the original dense version (140kB).

of non-volatile storage. Furthermore, it needs to observe that some of the FLASH memory space can be occupied by other routines (e.g. data pre-processing pipelines or operating systems). At last, it is not rare to have edge devices that run multi-task applications, which hence needing to store multiple ConvNet models, one for each different task. Besides filter pruning and quantization, as mentioned above, there is another technique very popular to compress ConvNet model size: weight pruning or sparsification. Sparse training used in combination with 8-bit quantization and lossless encoding is very effective to reduce the memory footprint of a generic ConvNet. The main step to reduce the storage needed to store the model parameters is the encoding, which takes advantage of the sparse and quantized array, as largely explained in Chapter 2. It is well known by literature how sparse training is very effective but still fragile. As a rule of thumb, the sparsity is proportional to the compression rate. However, under stringent constraints, i.e. a few tens of KBs, the technique becomes harder to manage as the ConvNet tolerance to the sparse format collapses quickly. This phenomenon can be clearly seen in Figure 4.2, which shows an example of standard iterative weight pruning applied on 9-layer ResNet ConvNet trained on CIFAR-10. The Blue dotted line is the dense baseline, the red line groups the sparse solutions for different memory constraints. Each star has a different memory target. It is easy to see how there is a tolerance boundary close to 90% percent of sparsity (around 48kB of storage): over this limit, the model suffers sudden and unrecoverable accuracy losses. This sudden drop of the accuracy loss is hard to manage and to predict, making state-of-the-art weight pruning not very suitable for compression under stringent memory constraints. This issue

poses a new challenge: *how to manage sparsity to increase compression yet preserving accuracy?*. Our work comes exactly in this direction: *EAST* (Encoding-Aware Sparse Training) is a novel optimization strategy able too boost compression for stringent storage-constraints. It is intuitive that at the same compression rate, would be preferable a less sparse ConvNet, as being more accurate. This can be done by focusing on the zero placement in the model arrays. The zero placement in sparse arrays is a serious issue for an accuracy-driven compression pipeline, where the encoding benefit constraints are underestimated. EAST aims to exploit this feature, it prunes blocks of neighboring weights rather than pruning single connections. In this way, a proper encoding scheme is able to reduce the sparsity needed to meet the memory target. EAST implements a sparse training procedure able to drive model compression by an encoding perspective. The main objective is to meet a given target FLASH memory, with minimum sparsity. This is done by forcing encoding benefits avoiding performance degradations due to extreme sparsity percentage. The crucial tool of EAST is an adaptive pruning strategy able to modulate the block size adapts to the memory constraint, minimizing the amount of sparsity needed. To encode the sparse arrays EAST uses the LZ4 [136] compression algorithm, which is a very suitable solution for MCUs as it guarantees extremely fast decompression and it requires a lightweight routine of few bytes of memory. However, EAST working is independent of the compression algorithm, as other encoding schemes can be seamlessly applied. We tested EAST on Arm Cortex-M4 MCU using a state-of-the-art 9-layer ResNet trained on the CIFAR-10 dataset. State-of-the-art comparison respect a standard weight pruning compression (using the same LZ4 scheme) proves EAST can achieve higher accuracy (up to 8.73%) when the available memory is very limited (12KB of FLASH).

4.2 Optimality Assessment of Memory-Bounded ConvNets

The total memory footprint M_c need to store a generic ConvNet is the sum of all its parameters N_p, namely weights and bias, each one represent with a fixed bit-width b. Given a target memory, M_t, is the actual on-chip memory that can be allocated. It needs to be noted that M_t can be lower than the physical memory available on-chip as other applications may run in the background. A memory-bounded ConvNet is a compressed version of the original floating-point ConvNet such that $M_c \leq M_t$ and the accuracy loss $\mathcal{L}$ is minimized. For different values of M_t there exists a set $\mathcal{P}$ of pairs $\{N_p, b\}$ that match M_t. Within $\mathcal{P}$, a pair $\{N_p^{\text{opt}}, b^{\text{opt}}\}$ is said *optimal* if it minimizes $\mathcal{L}$; a pair $\{N_p', b'\}$ is said *hardware-compliant* if can be ported on a physical device. b^{opt} can be of any integer value, while b' must be supported by a proper instruction set, i.e. $b' \in \{8, 16\}$ for our case study. An optimal pair is hardware-compliant if b^{opt} turns out to be 8- or 16-bit.

To assess the memory-accuracy solution space we developed an evaluation framework able to (*i*) compress the memory-bounded ConvNet combining pruning (to reduce N_p) and quantization (to reduce b), and to (*ii*) emulate and deploy the compressed ConvNets.

4.2.1 PaQ: Prune and Quantize

Figure 4.3 shows the proposed framework. The toolchain starts with a floating-point ConvNet (FP) trained within a standard dense training and it provides as output (*i*) the accuracy assessment of the compressed ConvNets that match the memory constraint (i.e. those with $\{N_p, b\} \in \mathcal{P}$) and (*ii*) the `.c` description of the compressed ConvNets that are hardware-compliant ($\{N_p', b'\} \in \mathcal{P}$). The `.c` is assembled using the CMSIS-NN [112] by ARM, then compiled and flashed on the target device. To assess those compressed models not able to meet the memory target (i.e. those with $\{N_p, b\} \in \mathcal{P}$, $b \neq b'$) we used an in-house fixed-point emulator; the same emulator is used to drive the fine-tuning stages (more details in Sec. 4.2.1). The optimization kernel, called Prune and Quantize (**PaQ** hereafter), consists of two main stages: (*i*) quantization-aware (**Q-aware**) filter pruning; (*ii*) model quantization using a FX representation of b bits (**b-Quantization**). Both stages receive the parameter b as an internal constraint. We used the same uniform bit-width across all the model layers. This latter aspect has an impact on the compression speed, that is, quantization removes information at a faster pace. It is intuitive that removing a single filter on a layer is less intrusive than reducing the bit-width of all the weights of all the layers. A more interesting aspect is that there exists a circular dependence between pruning and quantization which frustrates the optimization. In fact, the value of b and the number of removable filters are connected.

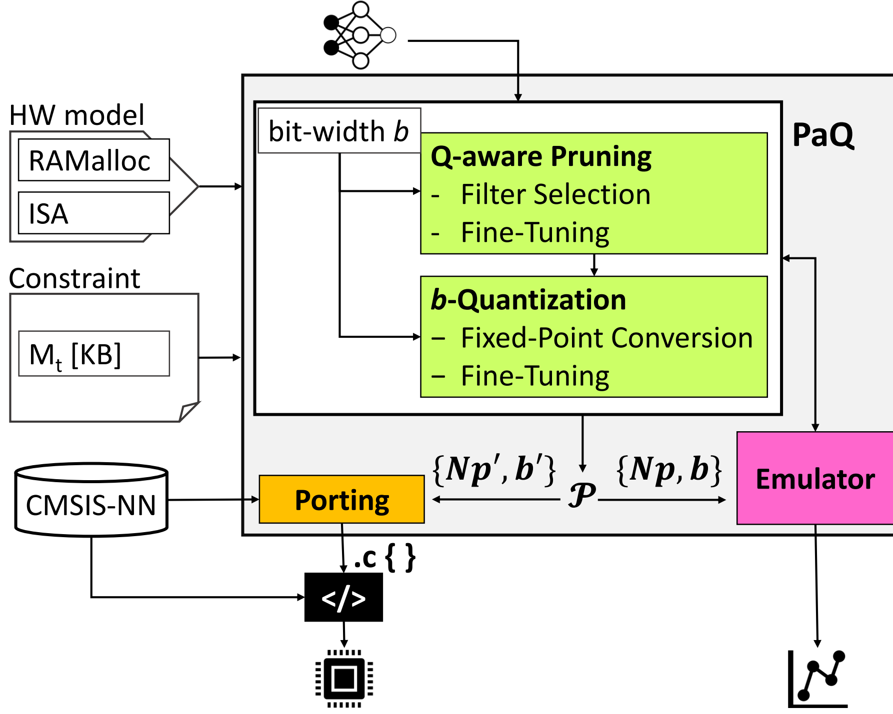


Figure 4.3: Framework overview.

The physical RAM required in the inference processing is estimated through a model file containing the memory allocation strategy of the target architecture (*RA-Malloc*). The proposed framework has been designed to work for any commercial MCU, as long as the HW-model and the neural-kernels library are available.

Memory Model

During each forward pass, a fixed quantity of on-chip RAM is allocated to process the input samples through the ConvNet layers, this amount depends on the implementation of the compute kernels, which in turn are directly correlated with the underlying hardware architecture. Our memory model is designed on top of the open-source CMSIS-NN library Cortex-M cores (v.5.4.0) by ARM [112], but can be extended to operate with other libraries and/or architectures. To describe the model we refer just to the convolutional layers, both because they are the most expensive in terms of memory utilization and considering the most recent ConvNets limit the number of fully connected layers to reduce the memory accesses. The Cortex-M family of MCUs are equipped with a flash memory used to permanently store the parameters of the ConvNet. At run-time, both weights and biases are loaded in blocks in RAM, in particular, in a portion referred to the *Weight Buffer* (WB). The partial results of each computation are temporarily stored in the remaining part of the RAM, which is taken by the *Activation Buffer* (AB).

Within a ConvNet, the structure of layers differs sensitively for the number of channels and for the size of each kernel, hence each layer requires a different quantity of memory to be processed. This value is proportional to the size of its input and output tensors. As ConvNets are processed layer-by-layer, AB is time-shared and its size is defined by the largest layer.

At last, during the processing of convolution operations, part of memory is used to store temporarily needed data, which are included in our memory model. These operations are implemented as generic matrix multiplications by the CMSIS-NN, where the multidimensional tensors are converted to 2-D array, namely Toeplitz matrix [199]. Then this matrix is processed by the *im2col*, which stores the results of the multiplication in a dedicated region of the RAM, the *im2col* buffer (I2CB). Similar to AB, the I2CB is time-shared among layers and its size is defined by the largest layer as well. To manage the inference stage on memory-bounded cores, a partial *im2col* routine is commonly adopted. To reduce I2CB at the cost of some performance overhead, these routines expand a selected portion of the input generating two columns of the Toeplitz matrix at a time.

The overall RAM footprint is then provided by the sum of the three buffers: $M_c = WB + AB + I2CB$. Equation 4.1 gives the analytical model for a b-quantized ConvNet composed of L layers:

$$M_c = b \times \left[N_p + \max_{i \in L}(I_i + O_i) + \max_{i \in L}\left(im2col_i\right) \right] \tag{4.1}$$

The first term (N_p) is the total number of parameters of the ConvNet and it refers to the WB buffer. For convolutional layers, the total number of weights is the product between the number of output channels, the number of input channels, and the size of kernels ($height \times width$), while the number of biases equals the number of output channels. In this work, the output channels are considered the convolutional filters. For fully connected layers, the number of weights is the product between the input and the output dimensions, while that of biases is the same as the dimension of the activation. The second term ($\max(I_i + O_i)$) refers to the AB buffer, where I_i and O_i are respectively the input and output sizes of the activations of the layers. At last, the ($\max\left(im2col_i\right)$) term is for the I2CB buffer. During the filter-by-filter processing by the *im2col* routine, the total sum of the memory needed for a convolutional layer is provided by the product of the three dimensions of a filter ($height \times width \times depth$), multiplied by a 2 (two columns of the Toeplitz matrix). Also, in this case, the max operator takes the largest contribution among all the layers. As a side note, the contribution due to I2CB is usually negligible, while the WB contribution is the most significant. At last, the AB strongly depends on the model topology; if the ConvNet is designed with small size kernels, like those adopted into embedded applications, AB contribution is not negligible (ranging from 15% to 30% the overall RAM).

Q-Aware Pruning

PaQ removes filters from the ConvNet until the target memory M_t is met. To select the correct number of filters to remove, the pruning stage needs to be aware of the bit-width b used for quantization, ad the final memory M_c depends on the numerical precision of the model. Removing one filter from the i-th layer simultaneously influences other several parameters of Equation 4.1: the cardinality of both the i-th and i+1-th convolutional layers (WB), the cardinality of the output activations of the i-th layer O_i (AB), the memory needed by the *im2col* to process the (i+1)-th layer (I2CB).

The pseudo-code of Algorithm 3 describes the details of the iterative procedure of Q-aware pruning. To sort the filters by the importance we adopted the ℓ_1-norm of its weights. This metric is a good importance estimator both for the low complexity (it is just computed on weights, avoiding the activations) and for the ability to identify the more redundant filters inside a layer [119], that are the ones who less affect the prediction accuracy. For these reasons, ℓ_1-norm is a good compromise between quality-of-results and complexity of the optimization loop. However, it needs to be noted that *PaQ* framework adapts also to other criteria of importance.

The loop iterates until the memory constraint M_t is met (line 2). The memory model introduced in the previous section is the main core of the memory estimation stage, in particular, it is embedded into the *RAMalloc* procedure. The estimation of the memory footprint is based on the physical bit-width b. However, to give to pruning a proper awareness of quantization the model is not quantized until reaching this stage.

Once the model is properly pruned, the *fine-tuning* stage is crucial to recover the accuracy loss (line 7). This stage consists of a set of re-training epochs (50 in our experimental set-up) during which the model re-adapts the weights to its new topology, using a standard error back-propagation scheme.

B-Quantization

After the Q-aware pruning, the model undergoes the actual quantization using a b-bit representation. As already motivated in Section 4, the choice fell upon the most hardware-friendly quantization: (i) symmetric scheme, (ii) linear intervals [165], (iii) per-layer power-of-two scaling. Adopting a per-layer radix-point scheme brings higher accuracy without performance overhead. Even though asymmetric quantization might provide more accurate results, it has significant overhead when the model is run on low-power IoT architectures, up to 20% [112]. Finally, a binary radix-point can be implemented with a simple bit-shift operation. The optimal choice is found through iterative optimization, different position of the radix-point are tried.

Also after the quantization process, there is fine-tuning stage to recover the

Algorithm 3: Q-aware pruning algorithm.

Input: ConvNet [FP-32], Target Memory M_t, Bit-width b
Output: Compressed ConvNet
1 M_c = RAMalloc(ConvNet[FP-32], b)
2 **while** $M_c > M_t$ **do**
3 Layer = Pick layer with lowest ℓ_1-norm
4 Filter = Pick filter of Layer with lowest ℓ_1-norm
5 Remove Filter
6 Update M_c
7 Fine-Tuning
8 **return** Compressed ConvNet

accuracy loss (totally or partially depending on the actual constraints). We implemented a custom fine-tuning stage iterated for 50 epochs: the forward-propagation is done with fake quantization (i.e., the weights are emulated to fixed-point representation), while the back-propagation kept the weights in floating-point full precision format to allow better re-adaption also with small updates. At last, at the end of each epoch, the weights are quantized through stochastic rounding.

To emulate fixed-point arithmetic on GPUs we developed an in-house emulator to leverage the fake-quantization method introduced in [64]. It consists of a software wrapper that converts activations and weights (stored in fixed-point) to the 32-bit floating-point; after being processed, results are converted back to fixed-point.

Porting and Emulation

Once compressed, the optimized ConvNet is translated in `.c` code using the custom kernels optimized for the target device. This work leverages the CMSIS-NN [112] library developed by ARM. It is a collection of optimized routines implementing the most common layers of deep neural networks and targeting the Cortex-M architecture. As already mentioned, the porting can be accomplished only for those bit widths and memory budgets that meet the hardware constraints ($\{N'_p, b'\}$). The framework provides emulation also for the other bit-width and memory constraints with the aim to estimate the distance between optimal and hardware-compliant solutions, which is one of the objectives of this work. The emulator is the same used within the **PaQ** flow.

4.2.2 Experimental Results

We used the proposed **PaQ**-based flow to explore the memory-accuracy space. The analysis has two main objectives: (i) assess the optimality of hardware-compliant

implementations, (ii) quantify the distance, in terms of accuracy, between these hardware-compliant solutions and the theoretical ones.

This section is organized as follows. First, we introduce the ConvNets and relative datasets used to benchmark the assessment. Second, we describe the hardware set-up, in particular the boards used as test-bench. Third, we present the collected results discussing the key findings. Finally, we provide additional insights to validate **PaQ** and justify the selected optimization strategies. For the sake of comprehension, we summarized all the notations used throughout the text in Table 4.1.

Notation	Description
M_c	Memory footprint of the ConvNet
M_t	Memory Target
N_p	Number of network parameters (weights and biases)
b	Bit-width (from $b_{\text{min}} = 2$ to $b_{\text{max}} = 16$, step 1-bit)
M_b	Memory footprint of the ConvNet (b-bit, w/o pruning)
$\mathcal{P}$	Set of pairs $\{N_p, b\}$ that matches M_t
$\mathcal{L}$	Top-1 accuracy loss
$\mathcal{L}_{\text{max}}$	Top-1 accuracy loss boundary ($= 0.5\%$)
$\mathcal{T}$	Pleateau area collecting the $\{N_p, b\}$ pairs s.t. $\mathcal{L} \leq \mathcal{L}_{\text{max}}$
P_x	Best-accuracy point
P_n	Pareto points in the memory-accuracy space ($n \in \mathbb{N}$)
PaQ-8	PaQ solutions (8-bit)
PaQ-16	PaQ solutions (16-bit)
Δ	Accuracy difference (optimal vs. HW-compliant)

Table 4.1: Table of abbreviations.

Benchmarks, Datasets and Training

To validate our technique on real-case scenarios of the IoT domain, we selected three different tasks: Image Classification (IC), Keyword Spotting (KWS), Facial Expression Recognition (FER). All of them find application in several domains, like healthcare, robotics, human-machine interface, and retail.

Each task is powered by a different ConvNet model which has been carefully selected among those that can be realistically deployed on IoT devices. For each task, we carefully selected a proper ConvNet model based on state-of-the-art. Table 4.2 reports the topology of the models together with the relative baseline (top-1 classification accuracy achieved using the original full precision model with no optimization). Results are consistent with those available in the literature. For both the training and inference stages, we used the popular deep learning framework

	IC		KWS		FER	
Dataset Input	CIFAR-10 [109] $3 \times 32 \times 32$		Speech Commands [208] $1 \times 32 \times 40$		FER2013 [16] $1 \times 48 \times 48$	
ConvNet Topology	Conv	(32,5,5)	Conv	(64,20,8)	Conv	(32,3,3)
	MaxPool	(3,3)	MaxPool	(1,3)	Conv	(32,3,3)
	Conv	(32,5,5)	Conv	(64,10,4)	Conv	(32,3,3)
	MaxPool	(3,3)	MaxPool	(1,1)	MaxPool	(2,2)
	Conv	(64,5,5)	FC	(32)	Conv	(64,3,3)
	MaxPool	(3,3)	FC	(128)	Conv	(64,3,3)
	FC	(10)	FC	(12)	Conv	(64,3,3)
					MaxPool	(2,2)
					Conv	(128,3,3)
					Conv	(128,3,3)
					Conv	(128,3,3)
					MaxPool	(2,2)
					FC	(7)
Top-1 Acc.	82.80%		86.75%		66.48%	

Table 4.2: Overview of the benchmark. Each model is composed by three types of layers: Convolutional (Conv) of shape (c_{out}, k_h, k_w), max-pooling (MaxPool) of shape (k_h, k_w), and fully-connected (FC) of shape (c_{out}). The height and the width of the kernels are defined as k_h and k_w, while the number of output channels is defined as c_{out}.

PyTorch [158] (version 0.4.1). For all three tasks we used the same training setup. Each model is trained for 150 epochs with Adam optimization [106], and batch-size of 128 samples. The learning rate starts from 0.001 and follows a linear decay scheduler of 0.1 every 50-epochs.

Board	Core	RAM	Flash	Frequency
NUCLEO-F412ZG [152]	Cortex-M4	256kB	1MB	100MHz
NUCLEO-F767ZI [154]	Cortex-M7	512kB	2MB	216MHz

Table 4.3: List of the development boards adopted to assess the compressed ConvNets.

Image Classification (IC). It is the extraction of information classes from a generic raw image. We used the popular *CIFAR-10* dataset [109]: it includes 60k 32×32 RGB images, divided in 10 different classes. The samples are split into 45k samples for training, 5k for validation, and 10k for testing. For this task,

we used a ConvNet model composed of three convolutional layers interleaved with max-pooling and one fully-connected layer. The ConvNet is taken from the Caffe framework [99], following the example showed in [112].

Keyword Spotting (KWS). A well-known application in the field of speech recognition, which is hard to deploy on low-power devices. However, considering the real-case IoT scenario, the KWS task is usually simplified to simple command detection (used as triggers), e.g. “Yes”, the task achieves an affordable level of complexity[1]. We used the popular Google Speech Commands Dataset [208], which is composed of 65k 1s-long audio samples collected during the repetition of 30 different words by thousands of different people. The goal is to recognize 10 specific keywords, i.e. “Yes”, “No”, “Up”, “Down”,“Left”, “Right”, “On”, “Off”, “Stop”, “Go”, out of the 30 available words. Samples that do not belong to the 10 categories are labeled as “unknown”. An additional “silence” class is made up of background noise samples (i.e. pink noise, white noise, and human-made sounds).

The samples are divided into training and test sets, respectively with 56196 and 7518 samples. The adopted ConvNet model is the *cnn-trad-fpool3*, used on the same task in [171]. The topology includes two convolutional layers, two max-pooling layers, and three fully-connected layers. The raw audio samples need a pre-processing stage to be classified by the ConvNet. In details, we followed the pipeline introduced in [171]: the recorded audio samples are converted to spectrogram samples of shape $time \times frequency = 32 \times 40$. No data augmentation has been used.

Facial Expression Recognition (FER). It is the understanding of the emotional state of people from their facial expressions. Quite popular in the field of visual reasoning, this task is very challenging as many face images might convey multiple emotions, hence it is difficult to isolate the correct label. We used the *Fer2013* dataset provided by the the Kaggle competition [16]. It includes 32297 48×48 grayscale facial images, divided into 7 categories: “Angry”, “Disgust”, “Fear”, “Happy”, “Sad”, “Surprise”, “Neutral”. The training set counts of 28708 samples, while the remaining 3589 are kept as the test set. We used a custom ConvNet architecture[2] composed of nine convolutional layers evenly spaced by three max-pooling layers and one fully-connected layer.

Hardware Specifications and Tools

The proposed framework is validated on two off-the-shelf boards powered with Cortex-M cores by ARM: NUCLEO-F412ZG [152] and NUCLEO-F767ZI [154]. The former is equipped with 256kB of RAM and 1MB of FLASH, while the latter

[1] https://www.tensorflow.org/tutorials/sequences/audio_recognition

[2] Inspired by https://github.com/JostineHo/mememoji

is equipped with 512kB of on-chip SRAM and 2MB of FLASH. Further details are reported in Table 4.3. For the compute kernels we used the CMSIS-NN library v.5.4.0 provided by ARM. The `.c` source file is compiled using the GNU Arm Embedded tool-chain (version 6.3.1). The **PaQ** tool is tuned for the ARM Cortex-M integer unit.

To validate the classification accuracy after the **PaQ** optimization, we used the emulator described in Section 4.2.1. The hardware-compliant ConvNets, i.e. the only ones able to be run on MCU, are validated directly on-board. Contrarily, to assess the solutions that cannot be flashed into the ARM cores we used the simulator. Figure 4.3 depicts an overview of this validation mechanism. Both training and simulated inference experiments were run on a GP-GPU workstation powered with a Titan GTX-1080 Ti by NVIDIA.

The *RAMalloc* memory model is cross-validated with the results provided by the `gcc` compiler (all the variables are statically allocated) and those returned by tracking the memory usage at run-time (feature available with the mbed-os operating system[3], version 5.11.0).

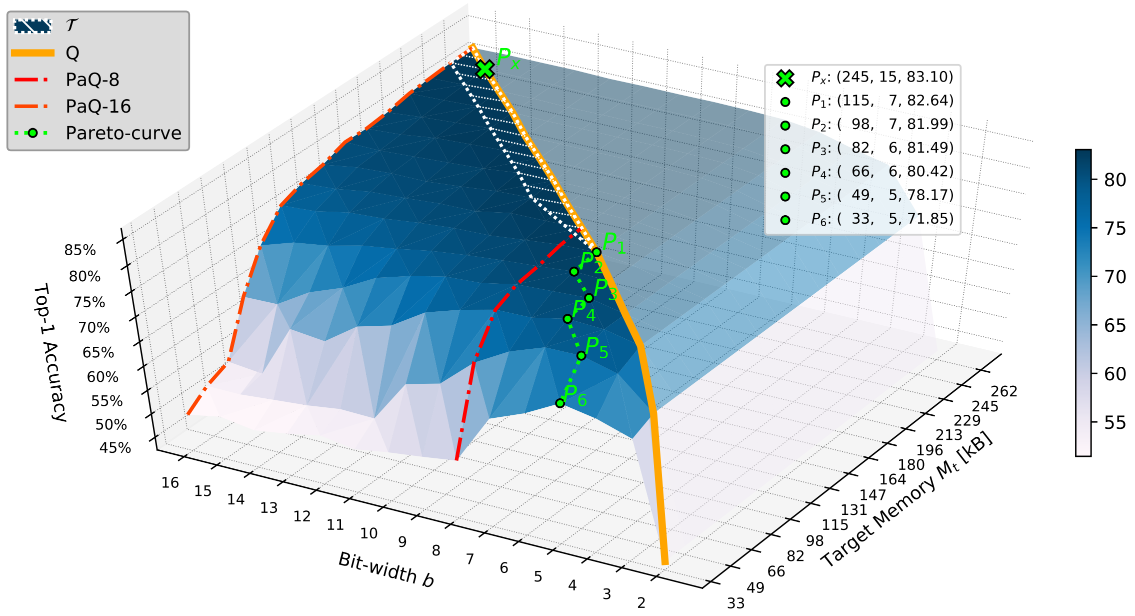
4.2.3 Across the Memory-Accuracy Space

The exploration is run for a discrete set of memory constraints, i.e. $M_t \in [M_{b_{\min}}, M_{b_{\max}}]$, $b_{\min}$=2, $b_{\max}$=16, step one bit; M_b refers to the memory footprint of the ConvNet quantized with b bits w/o any pruning (e.g. M_2 is the memory after a 2-bit quantization). For intermediate memory constraints, i.e. $M_t \in (M_{b_i}, M_{b_{i+1}})$, the accuracy is interpolated (more details in Section 4.2.3).

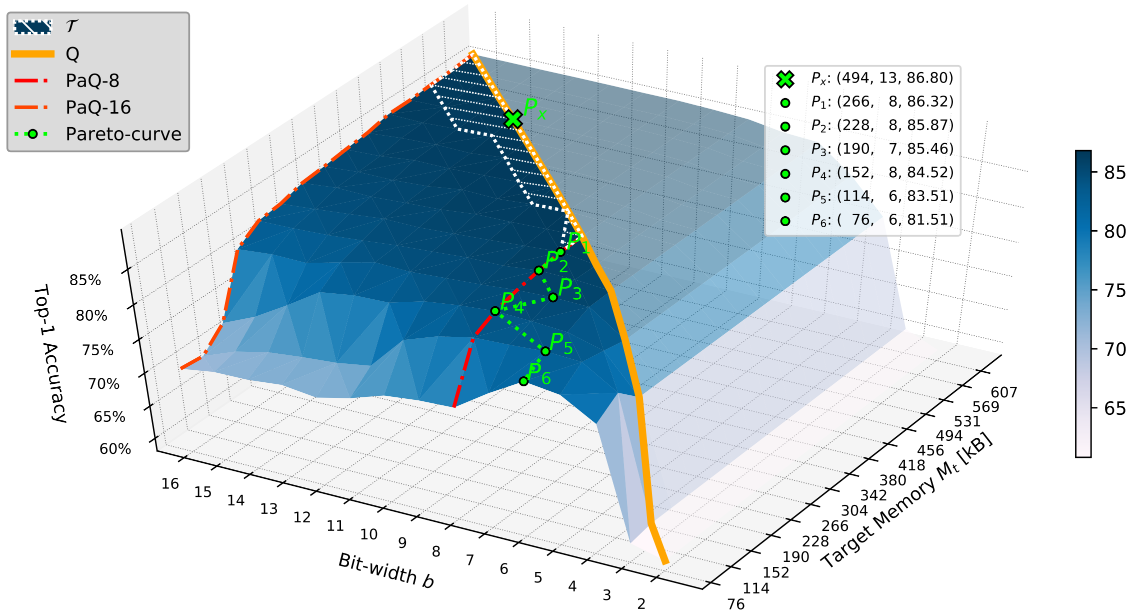
The collected results of the three applications are illustrated in Figures 4.4a, 4.4b and 4.4c. The surf plots show the top-1 accuracy for every solution point ($\{N_p, b\} \in \mathcal{P}$). The b-Quantization only solution (label Q), which is the only one without pruning, is highlighted in yellow. Over the yellow line, there is a transparent region with trivial solutions, in fact, all these implementations ($M_t > M_b$) are dominated by quantization. Since the Q-aware pruning skips the filter pruning as soon as the M_t is met, there might be memory-compliant solutions that belong to this line, e.g. in Figure4.4a the 2-bit quantization alone meets the memory constraint of 33kB.

Weakness of accuracy-driven optimizations. There is a plateau $\mathcal{T}$ (hatched area in the plot) where the accuracy gets very close to that of the original full precision model, namely pruning and quantization impact accuracy marginally. Without loss of generality, we assume that a pair $\{N_p, b\}$ belongs to $\mathcal{T}$ if the accuracy drop with respect to the best-accuracy point (P_x, marked with the green cross in the plots and reported in the first row of Table 4.4) is less or equal than 0.5%.

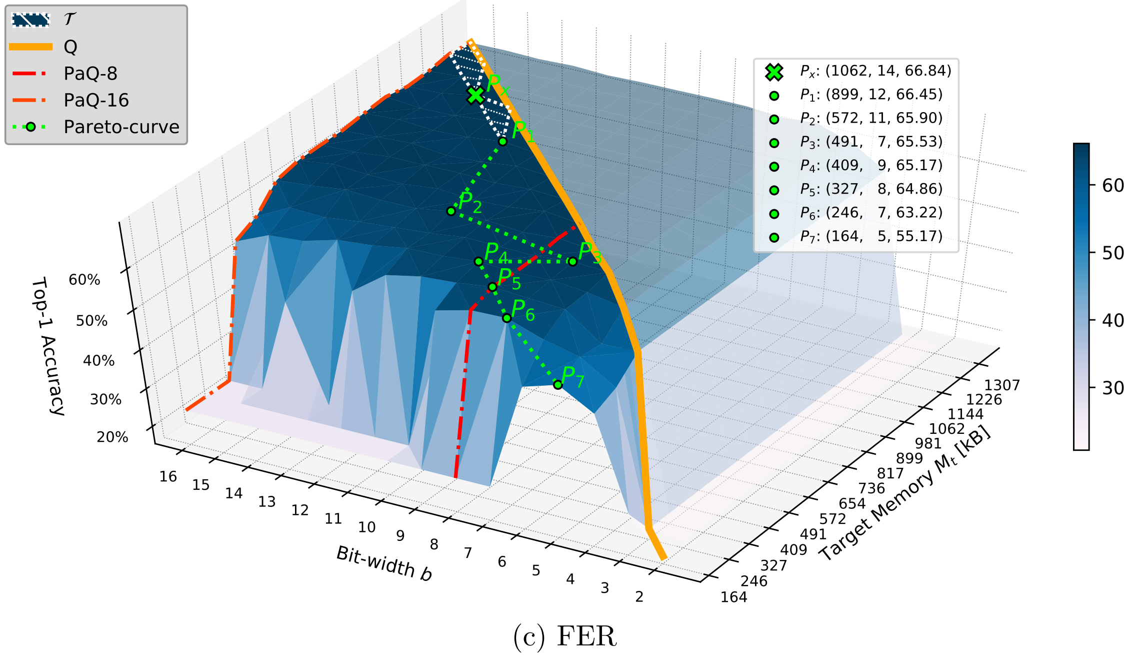
[3]https://os.mbed.com/blog/entry/Tracking-memory-usage-with-Mbed-OS/



(a) IC



(b) KWS



(c) FER

Figure 4.4: Solutions provided by **PaQ** flow showed in the memory-accuracy space for the three tasks under analysis (a) IC, (b) KWS, (c) FER. The green cross marker (P_x) shows the solution with higher accuracy, while the hatched area enclosed by the white dotted curve highlights the plateau region ($\mathcal{T}$), where the solutions have the accuracy loss $\mathcal{L} \leq 0.5$ w.r.t P_x. The yellow line (**Q**) indicates the solutions obtained applying only b-quantization. The red dash-dotted lines define the hardware-compliant solutions generated by **PaQ**, respectively using 8- (**PaQ-8**) and 16-bit **PaQ-16**; these are the implementations deployed on the physical device. The green dotted line connects the Pareto points (P) in the memory-accuracy space, i.e. all the solutions that have superior accuracy w.r.t. all the other points with the same target memory M_t. At last, all the absolute coordinates of the Pareto points and of P_x are collected on the right-side box, for each solution reporting (target memory, bit-width, top-1 accuracy). The right box collects the absolute coordinate of P_x and each Pareto point in the format (target memory, bit-width, top-1 accuracy).

The existence of $\mathcal{T}$ is nothing new as ConvNets are often redundant due to over-parametrization [67]. The area of $\mathcal{T}$ may depend on the complexity of the task or the network topology.

The accuracy-driven compression techniques proposed in the literature, e.g. [195], search for an unique combination of pruning and quantization which ensures the largest compression within a given accuracy loss $\mathcal{L}_{\max}$. Assuming a realistic constraint, e.g. $\mathcal{L}_{\max} = 0.5\%$, which is the same value used to define $\mathcal{T}$, the solution they return can be identified in our formulation as $\{N_p, b\} \in \mathcal{T}$ s.t. M_c is minimized. This solution represents the lower right corner of the plateau $\mathcal{T}$, denoted

		Pareto			**PaQ-8**			**PaQ-16**		
	M_t	P	b	Top-1	b	Top-1	Δ	b	Top-1	Δ
IC	245	P_x	15	83.10	8	82.85	0.25	16	82.86	0.24
	115	P_1	7	82.64	8	82.44	0.20	16	77.31	5.33
	98	P_2	7	81.99	8	81.40	0.59	16	72.52	9.47
	82	P_3	6	81.49	8	80.79	0.70	16	65.21	16.28
	66	P_4	6	80.42	8	78.85	1.57	16	54.85	25.57
	49	P_5	5	78.17	8	71.64	6.53	16	53.00	25.17
	33	P_6	5	71.85	8	54.68	17.17	16	50.00	21.85
KWS	494	P_x	13	86.80	8	86.38	0.42	16	86.20	0.60
	266	P_1	8	86.32	8	86.32	0.00	16	83.80	2.52
	228	P_2	8	85.87	8	85.87	0.00	16	83.48	2.39
	190	P_3	7	85.46	8	85.28	0.18	16	81.60	3.86
	152	P_4	8	84.52	8	84.52	0.00	16	73.11	11.41
	114	P_5	6	83.51	8	83.00	0.51	16	70.42	13.09
	76	P_6	6	81.51	8	75.16	6.35	16	70.78	10.73
FER	1062	P_x	14	66.84	8	65.34	1.50	16	65.23	1.61
	899	P_1	12	66.45	8	65.34	1.11	16	65.59	0.86
	572	P_2	11	65.90	8	65.48	0.42	16	63.47	2.43
	491	P_3	7	65.53	8	64.75	0.78	16	58.43	7.10
	409	P_4	9	65.17	8	64.61	0.56	16	55.92	9.25
	327	P_5	8	64.86	8	64.86	0.00	16	-	-
	246	P_6	7	63.22	8	63.03	0.19	16	-	-
	164	P_7	5	55.17	8	-	-	16	-	-

Table 4.4: Optimal vs. hardware-compliant solutions under different memory targets M_t. From left to right there are three main groups of columns: *Pareto*, **PaQ-8**, and **PaQ-16**. The first details the Pareto points P, the second details the hardware-compliant solutions provided by **PaQ** flow using 8-bit, and the third details the hardware-compliant solutions provided by **PaQ** flow using 16-bit. All these solutions are the same shown in the plots of Fig. 4.4. For the hardware-compliant solutions (**PaQ-8**, and **PaQ-16**) we reported also their accuracy distance (lower is better) to the optimal points in the column Δ. Solutions with too high accuracy losses ($\ll$50%) have not been reported.

with P_1 (second row of each benchmark in Table 4.4).

An accuracy-driven, memory-unconstrained optimization of this kind might return ConvNets that do not fit into the physical memory. Let's consider FER for instance, the optimal implementation would take 899kB of RAM using 12-bits, a configuration that is simply too large for our target devices. The focus of this

works is to explore the solutions placed in the *deep memory space*, that is the region below such theoretic optimum. One may argue that other meta-heuristics, like Bayesian Optimization, can be guided towards this region of interest by integrating the memory footprint in the cost function. That is true in general, but those methods perform better in optimization rather than fine exploration. Moreover, the multi-objective function may result biased by the importance weights adopted.

Memory-accuracy Pareto curve. There is a Pareto curve in the deep memory space, which is highlighted with a green dotted line in the plots. As already discussed, P_1 corresponds to the solution $\{N_p, b\}$ inside $\mathcal{T}$ with the smallest memory footprint. Instead, the remaining Pareto points lay outside $\mathcal{T}$ and represent those implementations able to meet lower memory constraints at the cost of higher accuracy losses, $\mathcal{L} > 0.5\%$. The existence of these points can be intuitive, but a quantitative analysis may reveal interesting trends. The exact values of target memory M_t and bit-width b are reported in Table 4.4 together with the top-1 accuracy they achieve. As the numbers suggest, for many configurations, the obtained accuracy gets very close to the best accuracy, yet ensuring substantial memory reduction. For instance: KWS shows a small accuracy drop of 1.34% (from 86.80% to 85.46%) with 62% of memory compression (from 494kB to 190kB); FER goes even better by showing 46% memory reduction (from 1062kB to 572kB) within a negligible accuracy loss of $< 1\%$ (from 66.84% to 65.90%). Similar conclusions can be inferred from the comparison among the other Pareto points.

Optimality of hardware-compliant solutions. A more interesting analysis concerns the distance, in terms of top-1 accuracy, between the implementations on the Pareto curve (theoretical solutions) and the hardware-compliant implementations (practical solutions), i.e. the pairs $\{N'_p, b'\}$ with $b' \in [8, 16]$ highlighted with the red dash-dotted curves in the plots (labels PaQ-8 and PaQ-16 respectively). The top-1 accuracy for PaQ-8 and PaQ-16 are given in Table 4.4, together with the distance from the Pareto curve (column Δ). The results show that PaQ-8 outperforms PaQ-16 (smaller Δ). There are only two exceptions, i.e. IC at $M_t =$ 245kB and FER at $M_t =$ 899kB, yet with a mere distance (0.25% in the worst case). The actual reason is that under the same memory budget, the 8-bit model has more remaining filters, hence the accuracy of 8-bit model is higher than the corresponding 16-bit one. In other words, the 8-bit models stop pruning earlier than 16-bit. This can also be proved by looking at numbers collected in Table 4.4, FER benchmark under a memory constraint $M_t =$ 327kB: the 16-bit model is so highly pruned that the accuracy falls down to impractical values, while the 8-bit model meets the memory constraint with fewer filters pruned and hence lower accuracy loss. The key insight is that a bit-width below the 8-bit mark is needed just for very tight constraints. For instance, KWS under $M_t =$ 76kB memory constraint, where the PaQ-8 implementation shows $\Delta \geq 1\%$, or FER under $M_t =$ 164kB memory constraint, where PaQ-8 do not provide reasonable accuracy loss. The conclusion is that arbitrary bit-widths are really needed just in few specific cases and the

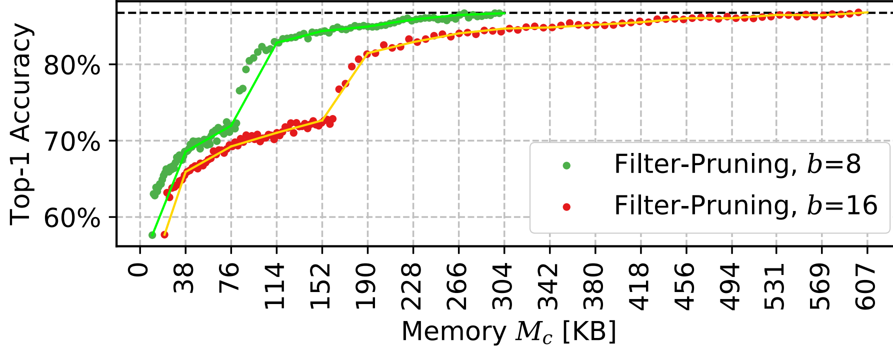


Figure 4.5: Memory footprint vs. Top-1 accuracy for KWS.

adoption of specialized architectures needs to be assessed carefully.

Validation of PaQ

Efficacy of the proposed memory-driven compression. As described in Algorithm 3, the proposed version of filter pruning is memory-driven, in fact, the optimization loop stops the filter removing as soon as the memory constraint is met. To motivate this stopping criterion, we provide the analysis of a pruning strategy where the constraint is given in a direct form, i.e. *number of filters to be pruned*; once pruned, the models are quantized and then fine-tuned to recover the accuracy loss. Then, the use of the number of filters as a control knob enables to span of the entire memory range. Figure 4.5 shows the results for KWS; the plot collects the achieved both with 8- and 16-bit. It is interesting to observe how the lines follow a pseudo-monotone trend: compressed memory accuracy decreases together. Negligible ripples are due to the noise introduced by fine-tuning. These results fully justify our choice: to stop the search as soon as the constraint M_t is met gives the highest accuracy for that specific M_t. Despite Figure 4.5 shows just one example for the sake of readability, the same trend is clearly visible for every bit-width used in our experiments. Furthermore, the linear interpolation adopted to estimate the accuracy when $M_t \in (M_{b_i}, M_{b_{i+1}})$ is also validated. Indeed, the plot shows that accuracy is a piece-wise linear function of memory. The same considerations hold for the other ConvNet benchmarks.

On the scalability of the proposed hardware-driven optimization. The adopted **PaQ** scheme is hardware-friendly, as the memory compression automatically improves latency too. We refer to this kind of scheme as *latency proportional*. Figure 4.6 shows the average latency for one feed-forward pass of the ConvNet used in KWS. The analysis is conducted under different memory constraints (the same reported in Table 4.4).

As PaQ-8 dominates PaQ-16, as described in the previous section, we reported

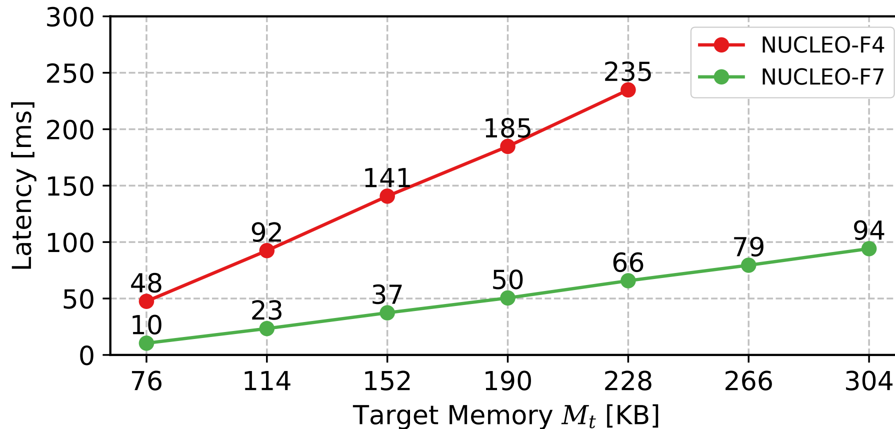


Figure 4.6: Average inference time per sample of PaQ-8 solutions on KWS.

the latency results just for 8-bit quantized ConvNets. The execution time is measured using the timer API provided by the mbed-os operating system and averaged over the entire test set. The experiments were run on both the boards reported in Table 4.3, labeled as NUCLEO-F4 and NUCLEO-F7 for brevity. The NUCLEO-F4 board has a maximum RAM of 256kB, therefore larger models cannot be deployed.

Adopting pruning and quantization schemes that preserve the regularity on the ConvNet topology is the key to achieve a direct proportionality between inference time and memory footprint. The choices implemented in the proposed framework go in this direction as they have been conceived (i) to cut the number of memory accesses, (ii) to alleviate the cost of the *im2col* procedure, and (iii) to reduce the number of operations as the memory footprint gets smaller. The result is the linearity shown in the plot. The same trend holds for the other benchmarks.

Execution Time. The **PaQ** flow takes a few minutes for each fine-tuning stage (50 epochs each). The actual execution time may vary depending on the complexity of the ConvNet and the memory constraint. The worst case is the largest benchmark (FER): 25 minutes on average for each $\{M_t, b\}$ pair, 80% spent for the fine-tuning stages. A significant reduction can be achieved limiting the number of retraining epochs: early stopping policies may be adopted to solve this issue, similarly as it has been done in other works to prevent over-fitting [164] or accelerate the training stage [6]. While the speed-up of the **PaQ** flow is out of the scope of this work, Table 4.5 supports our claim showing the number of fine-tuning epochs after which **PaQ** is already able to reach the highest top-1 accuracy.

Collected numbers refer to the average over all the pairs $\{M_t, b\}$ of the exploration space. For the three benchmarks, both pruning and quantization converge much earlier than the 50-epoch threshold we set for safety, revealing the potential margins.

	Q-aware Pruning	**b-Quantization**
IC	30.7	22.1
KWS	27.7	15.4
FER	18.7	13.5

Table 4.5: Training time needed (expressed as the number of epochs) to reach the baseline accuracy for each task.

4.2.4 Conclusions

The overall outcome of the assessment enables three main achievements. First, **PaQ** demonstrates that the implementation of practical ConvNets is not just accuracy-driven, but instead, the actual memory constraint plays a crucial role. Second, our analysis enumerates the optimal configurations in the memory-accuracy solution space when the memory target is very very tight. Third, **PaQ** quantifies the actual distance between optimal (theoretical) configurations and the practical ones, which are the closest implementations deployable on low-power MCUs.

4.3 Arbitrary Bit-width ConvNets on IoT MCUs

Among the possible compression strategies, n-ary fixed-point quantization has proven to be very effective in reducing both computational effort and memory footprint, with high resilience to accuracy loss. However, as described in Section 2.3, its use requires custom components and special memory allocation strategies, which are not available and burdensome to implement on low-power MCUs. To bridge this gap, we proposed Virtual Quantization (**VQ**), a hardware-friendly compression method that enables equivalent n-ary ConvNets on general-purpose instruction-set architectures.

Virtual Quantization

The aim of **VQ** is to devise a ConvNet model tuned for a target bit-width H s.t.: (i) the resulting memory footprint is the same that would have been obtained through a classical n-bit quantization, being $n < H$; (ii) the classification accuracy is larger or equal than that of the theoretic n-bit model. Given H as the bit-width supported by the instruction set of the target core, the model obtained with **VQ** does emulate the n-ary model while preserving hardware compliance.

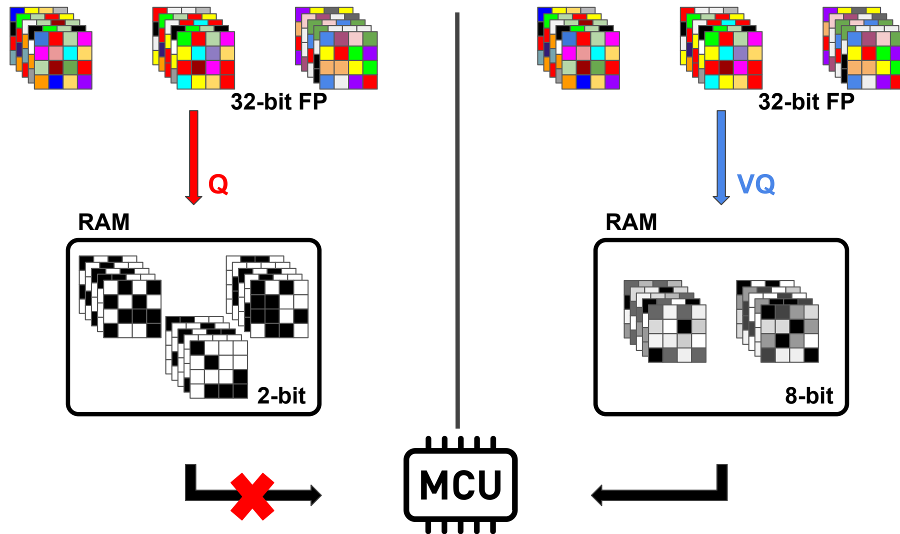


Figure 4.7: A pictorial representation of the comparison between the standard n-ary Quantization (left) and our proposed method, Virtual Quantization (right). The different depths of colors refer to the bit-width.

Flow Overview

VQ is implemented through the optimization flow depicted in Figure 4.8. The framework is fed with a floating-point ConvNet model (FP) trained with any standard deep-learning library (e.g. PyTorch, TensorFlow); it generates a `.c` description of the equivalent n-bit quantized fixed-point model (FX-H), with n the equivalent bit-width provided as input. The high-level description of the network is translated into a low-level code using a neural network library (NN library) optimized for the target hardware (deployment stage in Figure 4.8). In this work, we adopted the CMSIS-NN by ARM for the Cortex-M architecture [112]. The CMSIS-NN is a collection of optimized neural kernels which cover the most common operators: convolutions, fully-connected, activation and, pooling functions. The final output is a `.c` file that can be compiled and flashed on the target MCU. A memory model file describes the memory allocation strategy used to estimate the physical RAM needed at inference time. It is worth emphasizing that the RAM usage drives the compression; this is a distinctive feature w.r.t. classical accuracy-driven compression methods.

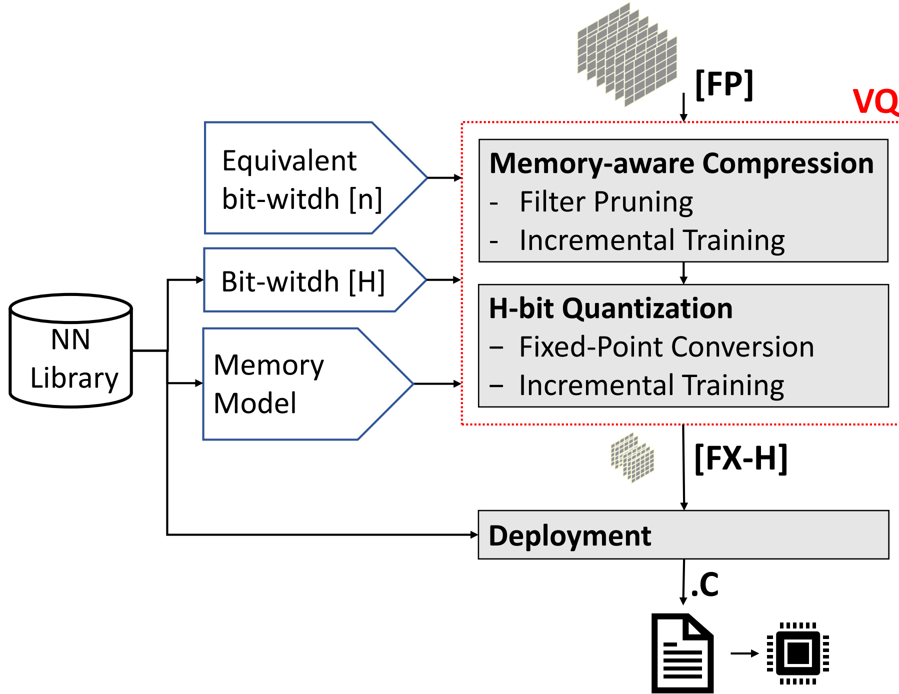


Figure 4.8: The Virtual Quantization flow.

The core of **VQ** consists of two main stages: (i) memory-aware compression; (ii) H-bit model quantization, with H as the bit-width supported by the instruction-set of the target core. As long as both the memory model and the neural network library are available, the **VQ** flow does apply to any micro-controller. Since the

target of our work is the ARM Cortex-M architecture, we built a memory model for the CMSIS-NN library and we assume $H = 8$. The next sections give a detailed overview of the **VQ** steps, with a preliminary description of the memory allocation model.

Memory Model

Typical state-of-the-art ConvNets are directed acyclic graphs whose scheduling is input-independent, hence the memory allocation can be done statically. One can precisely compute the memory footprint just by knowing the network topology and the linked neural library (the CMSIS-NN by ARM in this work). The data structures needed for the feed-forward execution of neural networks include: (*i*) a buffer storing the network parameters (weights and biases); (*ii*) a buffer for the input and output features; (*iii*) a buffer storing partial data used by neural networks routines. Since low-power MCUs, e.g. Cortex-M, have a very simple memory hierarchy, all the three buffers reside in RAM. The overall RAM space is thereby computed as the sum of the three contributions [112]. For what concerns the ConvNet parameters (*i*), they are permanently stored in the flash memory and then block-loaded in RAM at run-time thus avoiding the overhead of accessing the flash memory. Regarding the feature (*ii*) and internal (*iii*) buffers, the ConvNet layers are executed sequentially, therefore the corresponding memory can be time-shared between different layers.

The memory model is obtained from the GNU linker [51] and cross-validated with the statistics collected at run-time (we installed the lightweight mbed-os operating system for this purpose, v. 5.9.7). To estimate the memory footprint of the theoretic n-bit model, we extended the embedded memory model to arbitrary bit-widths assuming an ideal word size equal to n. To notice that this is a theoretic model as in real hardware the minimum word is usually greater, i.e. $H > n$; for instance, $H = 8$ in the Cortex-M architecture.

4.3.1 Memory-Aware Compression

A ConvNet with a fixed number of layers has two potential sources of redundancy: (*i*) the number of parameters within each layer; (*ii*) the arithmetic precision of the weights within each filter. The key observation over which **VQ** is built is that standard n-bit quantization operates on the second term only. Meeting a tight memory constraint would, therefore, require a bit-width n too small (usually much smaller than the minimum bit-width H made available by common HW). To overcome this issue, **VQ** implements a layer-wise compression which is based on a memory-aware filter pruning.

The iterative procedure of the memory-aware compression is described in the pseudo-code of Algorithm 4. At each iteration, the least important filter from the

Algorithm 4: Memory-aware compression algorithm.

Input: FP-Model, Bit-width H, Equivalent bit-width n
Output: Compressed Model
1 Target Memory = Memory of FP-Model at n-bits)
2 Current Memory = Memory of FP-Model at H-bits)
3 **while** *Current Memory* $>$ *Target Memory* **do**
4 Layer ranking
5 Pick less important layer
6 Filter ranking
7 Remove less important fitler
8 Update Current Memory
9 Incremental training
10 **return** Compressed Model

least important layer (lines 4–8) is dropped. As a *ranking* criterion, we used the sum of the absolute weights, i.e. the $\ell 1$-norm of the parameters. Weights with lower $\ell 1$-norm have less impact on the output features [119]. The loop iterates until the memory constraint is met (line 3). The memory estimation is run using the memory model introduced in the previous section (lines 1–2). As already discussed, it accounts for all the data structures used at the inference stage, not only the network parameters. It is worth emphasizing that the memory footprint is estimated depending on the physical bit-width of the target hardware, i.e. H, but the model is not quantized yet at this stage. This gives to the compression stage the proper awareness of quantization.

The model compression might degrade the quality of results. To recover the accuracy loss, we leveraged an incremental training procedure (line 9).

H-bit Quantization

After memory-aware compression via pruning, the model undergoes the actual quantization to H-bits. The CMSIS-NN library offers fixed-point neural kernels with a per-layer dynamic scheme based on power-of-two scaling. Adopting a per-layer radix-point brings better results as different layers show different dynamic ranges. Even though more complex scaling techniques, e.g. asymmetric quantization, might result more accurate, they might increase the execution time when the ConvNet is deployed on low-power MCUs, up to 20% according to [112]. The accuracy drop induced by quantization can be recovered (totally or partially depending on the actual constraints) using an incremental re-training procedure. The latter has the following main characteristics: the forward-propagation is run with fixed-point emulation; during back-propagation weights are kept in a floating-point format thus to allow small weight updates; weights are quantized at every epoch using

stochastic rounding. In order to emulate fixed-point arithmetic on general-purpose GPUs, we also implemented an in-house tool that leverages the fake-quantization method introduced in [63]. It consists of a software wrapper that converts activations and weights (stored in fixed-point) to the 32-bit floating-point; after being processed, results are converted back to fixed-point.

4.3.2 Experimental Results

Benchmarks, Datasets and Training

We tested the **VQ** framework on three popular tasks: Image Classification (IC), Keyword Spotting (KWS), Facial Expression Recognition (FER). Different lightweight ConvNets suited for tiny cores are deployed for each task; details reported in Table 4.6. Such ConvNets are trained for 150-epochs in PyTorch (version 0.4.1) using Adam optimization [106] (learning rate $1e-3$, linear decay 0.1 every 50-epochs, batch-size 128).

IC		**KWS**		**FER**	
CIFAR-10 [109] Input: $3 \times 32 \times 32$		Speech Commands [208] Input: $1 \times 32 \times 40$		FER2013 [16] Input: $1 \times 48 \times 48$	
Conv2d	(32,5,5)	Conv2d	(64,20,8)	Conv2d	(32,3,3)
MaxPool2d	(3,3)	MaxPool2d	(1,3)	Conv2d	(32,3,3)
Conv2d	(32,5,5)	Conv2d	(64,10,4)	Conv2d	(32,3,3)
MaxPool2d	(3,3)	MaxPool2d	(1,1)	MaxPool2d	(2,2)
Conv2d	(64,5,5)	FC	(32)	Conv2d	(64,3,3)
MaxPool2d	(3,3)	FC	(128)	Conv2d	(64,3,3)
FC	(10)	FC	(12)	Conv2d	(64,3,3)
				MaxPool2d	(2,2)
				Conv2d	(128,3,3)
				Conv2d	(128,3,3)
				Conv2d	(128,3,3)
				MaxPool2d	(2,2)
				FC	(7)

Table 4.6: Summary of the ConvNets used to validate VQ. Each model is composed by three types of layers: Convolutional (Conv) of shape (c_{out}, k_h, k_w), max-pooling (MaxPool) of shape (k_h, k_w), and fully-connected (FC) of shape (c_{out}). The height and the width of the kernels are defined as k_h and k_w, while the number of output channels is defined as c_{out}.

For **IC** we used the ConvNet delivered within the Caffe framework [99] according

to the experimental set-up reported in [112]. The data-set is the popular *CIFAR-10* [109], made up of 60k 32×32 RGB images labeled with 10-classes. Concerning **KWS**, we followed the experimental procedure introduced in [171], which makes use of the *cnn-trad-fpool3* ConvNet [171] to classify 10 keywords belonging to the Speech Command Dataset [208]. The training-set and test-set data are composed of 56196 and 7518 spectrograms respectively ($time \times frequency = 32 \times 40$). For the **FER** task we resorted to a VGG-like ConvNet which recognizes the facial emotion dataset provided by [16]. The dataset has 48×48 grayscale facial images classified by 7 labels; training and test set consist of 28708 and 3589 instances respectively.

Deployment and Emulation

We validated the **VQ** framework (Figure 4.8) for the Cortex-M family by ARM. Tests were run on the NUCLEO-F767ZI board by ST Microelectronics using the CMSIS-NN library v.5.4.0 provided by ARM. The GNU Arm Embedded tool-chain (version 7.3.1) was used to compile the `.c` level model. In order to emulate the n-bit quantization (for which there's no HW available) and to ensure a fair comparison with the **VQ** method, the inference accuracy of the three applications is measured through the fake-quantization tool mentioned in Section 10. Such a tool is made run on a GPU workstation powered with a Titan GTX-1080 Ti by NVIDIA and it offers several settings that adapt to different fixed-point HW units. For what concerns this work, the tool is tuned for the ARM Cortex-M integer unit. Extensive emulation runs show fake-quantization achieves 100% match with the results computed on the actual boards.

Results

Table 4.7 collects the results obtained with a standard n-bit quantization (**Q**) where models are scaled to an arbitrary fixed-point (FX) bit-width n. For a fair comparison, the adopted **Q** scheme consists of the same procedure deployed in the **VQ** flow during the H-bit Quantization stage. For each benchmark, the first row collects the classification accuracy of the original 32-bit floating-point model (FP). The next seven rows quantify the figures of merit of the n-bit quantization with $n \in [2-8]$. The column *RAM* shows the n-th model memory footprint computed with the memory model introduced in Section 4.3 (inline with results in [112]). As an additional piece of information, the *Core* column reports the smallest ARM Cortex-M with enough RAM to host and run the ConvNet. The *Top-1* column collects the top-1 accuracy achieved by each model.

As demonstrated in previous works (e.g. [165]), 8-bit quantization reaches almost the same accuracy of the original FP model. This is also confirmed by our experiments (the worst-case loss is 1.14% for FER). Lower bit-widths show a substantial degradation of quality: e.g. with $3-bit$ the accuracy loss ranges from

Task	**D-Type**	n**-bit**	**RAM** (KB)	**Core**	**Top-1** (%)
IC	FP	32	n/n	None	82.80
	FX	8	131	M4	82.85
		7	115	M3	82.64
		6	98	M3	81.79
		5	82	M3	80.05
		4	66	M3	78.60
		3	49	M3	70.24
		2	33	M3	44.14
KWS	FP	32	n/n	None	86.75
	FX	8	304	M7	86.38
		7	266	M7	85.68
		6	228	M4	85.41
		5	190	M4	81.90
		4	152	M4	77.19
		3	114	M3	61.70
		2	76	M3	9.07
FER	FP	32	n/n	None	66.48
	FX	8	654	None	65.34
		7	572	None	64.59
		6	491	M7	65.03
		5	409	M7	62.41
		4	327	M7	57.65
		3	246	M4	18.22
		2	164	M4	17.44

Table 4.7: VQ performance obtained on the three benchmarks under analysis.

12.56% (for IC) to 48.26% (for FER). The results confirm that weight quantization is an effective compression strategy. For instance, for FER, the 6-bit model would fit in cores with 512kB RAM (e.g. the M7) still guaranteeing a reasonable accuracy loss (1.45%). This analysis does not consider the lack of proper hardware architectures to process workloads with less than 8-bit. As already discussed, there are alternative software patches, but they proved to be very impractical [170].

Figure 4.9 highlights the key results obtained with the proposed **VQ** as it gives a fair comparison w.r.t. n-bit quantization **Q**, our baseline. For the sake of completeness, **VQ** is also compared against a classical pruning methodology (label **P**) whose details are introduced later. The bars show the top-1 accuracy loss, which is defined as the difference between the top-1 accuracy of the original FP model

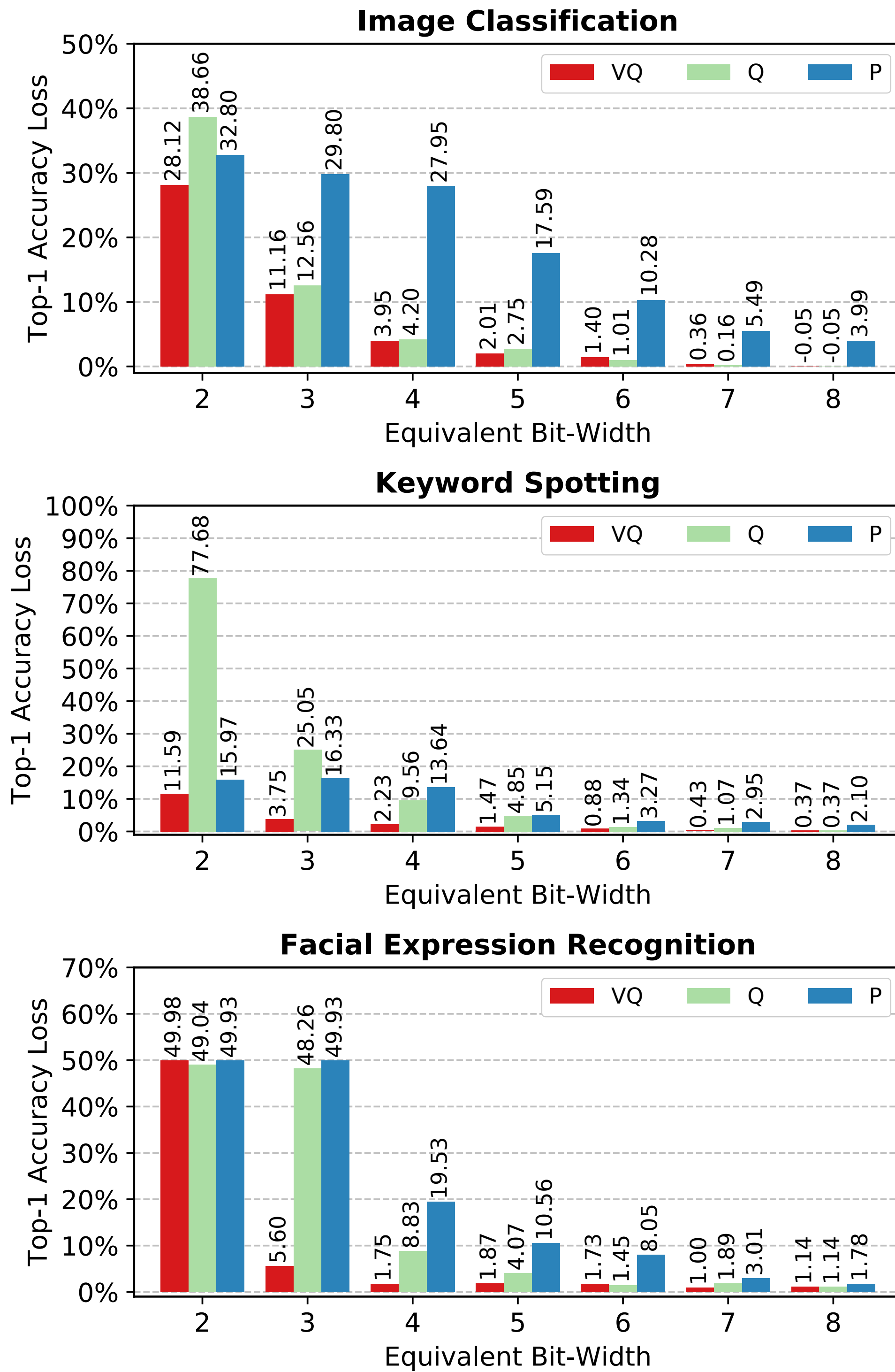


Figure 4.9: Top-1 accuracy loss in the three different applications (lower is better). The baseline accuracies are the same reported in Table 4.7.

(first row of each benchmark in Table 4.7) and that achieved after the compression. Each bit-width is associated with a different memory footprint, those obtained with the theoretic fixed-point n-bit quantization (i.e. those in Table 4.7). For the three applications and all the equivalent bit-widths, **VQ** converges to solutions that meet the same memory footprint of **Q**, yet achieving much lower losses (e.g. 7.08% less for FER at 4-bit). In other words, **VQ** makes pure theoretic n-ary quantization a practical solution even on general-purpose MCUs. Some exceptions might exist, like FER with 6-bit, where the **VQ** loss (1.73%) is larger than that of Q (1.45%). However, the gap is small (0.28%). The greedy nature of the heuristic and the statistic retraining stages are the first sources of such mismatch. Even more interesting, there are cases where **VQ** outperforms **Q** using fewer bits. For instance, KWS compressed with **VQ** set to 4-bits shows a loss (2.23%) that is smaller than that of **Q** with 5-bit (4.85%). The same holds for FER, which is the benchmark with the highest complexity and the largest memory footprint: **VQ** with 3 bits (5.60%) vs. **Q** with 4-bits (8.83%). That's due to the quantization-aware compression method integrated into **VQ**: it is aware of the underlying hardware constraints, hence it runs a well-balanced filter reduction.

One may argue that "memory-equivalence" can be achieved with any standard compression method (e.g. any standard accuracy-driven pruning) enhanced with the ability to meet a specific memory constraint. For such reason, Figure 4.9 also reports the result obtained with a "blind" pruning (label **P**) where filters are dropped one-by-one ($\ell 1$-norm as a ranking criterion) till the ConvNet fits the same RAM of that obtained with Q. Differently from **VQ**, the pruning strategy does not involve any quantization awareness, and filters are kept to maximum precision (16 bits for this case of study). The results clearly show that P meets the memory constraint of any equivalent bit-width but it gets worse in terms of accuracy. Moreover, its efficacy reduces with tight memory constraints. The worst-case is for FER with 3-bit, where the accuracy loss is 44.33% larger than that of **VQ**. The main reason is that pruning alone removes the filters at a too fast pace to reach the target memory footprint; this has dramatic effects on the classification accuracy.

As a final comment, we noticed that models which undergo the **VQ** flow get faster with the reduction of the equivalent bit-width. This is a direct effect of the compression method which, unlike quantization, reduces the number of memory accesses (both read and write) and the number of operations without requiring any extra code or special routine. Considering KWS for instance, with an accuracy loss of 1% (equivalent to 6-bit Q) the ConvNet model takes 25% less memory and is 1.4× faster.

4.3.3 Conclusions

We proposed a compression method for embedded ConvNets called Virtual Quantization **VQ**, whose main objective is to emulate a n-ary quantization on memory-bounded MCUs. We demonstrated that our technique allows to compress a ConvNet under the same memory target of a state-of-the-art n-ary quantization **VQ**, but with higher accuracy. This enables the deployment of the compressed ConvNet on general-purpose architectures without the need for custom components or special memory allocation strategies, which instead would be required using standard n-ary quantization. The feasibility of **VQ** has been proved on three different IoT real-case scenarios (IC, KWS, FER).

4.4 EAST: Encoding-Aware Sparse Training

Several recent works investigated aggressive optimization techniques for memory compression. Among them, quantization via fixed-point representation is now considered a mandatory step. The use of 8-bit integers is a common choice for general-purpose MCUs [112] as it reduces the memory footprint up to 4× w.r.t. 32-bit floating-point with no, or negligible, accuracy loss. Furthermore 8-bit quantization does not require custom kernels or particular allocation strategies, as described in Chapter 2. However, quantization alone might be not enough to fit stringent memory requirements. Sparse training via *weight pruning* [72][239] is an additional strategy that can improve the compression if combined with some encoding scheme and/or when quantization is jointly applied [68][195].

Inducing sparsity and quantization in ConvNets aids the compression by lossless encoding schemes as their weight tensors have repetitive and simple patterns, like long chains of zeros or repeated values. As a rule of thumb, the higher the sparsity, the larger the compression rate. However, as aforementioned in Section 4, there is a sparsity boundary over that this rule is difficult to exploit, as ConvNets experience a sudden, catastrophic degradation of accuracy. In fact, under stringent constraints, in the deep memory space, the level of sparsity that will let encoding schemes meet the constraint is extremely high, much higher than what ConvNets may tolerate.

The new challenge faced by this works is to achieve higher compression rates with lower sparsity to preserve accuracy. The rationale is simple, yet effective: find blocks (or groups) of adjacent weights to be pruned rather than pruning single connections. *EAST* (Encoding-Aware Sparse Training) implements a sparse training procedure based on *adaptive* pruning, minimizing the amount of sparsity needed to meet the constraint.

4.4.1 Storage-Aware Compression

Flow Overview

The optimization flow encompasses three stages: *(i) sparse training*, *(ii) quantization*, and *(iii) encoding*. The first trains the sparse network under a user-defined storage memory constraint. The second, *quantization*, reduces the arithmetic precision of the model to a given bit-width (8-bit in this work). The last, *encoding*, compresses the model size leveraging favorable patterns made available through the sparse training. The EAST strategy implements the first stage.

EAST

Encoding-Aware Pruning. Accuracy-driven weight pruning algorithms return tensors with sequences of zeros much longer than in the original dense model. Although this helps to increase the compression rate of the encoding scheme, there

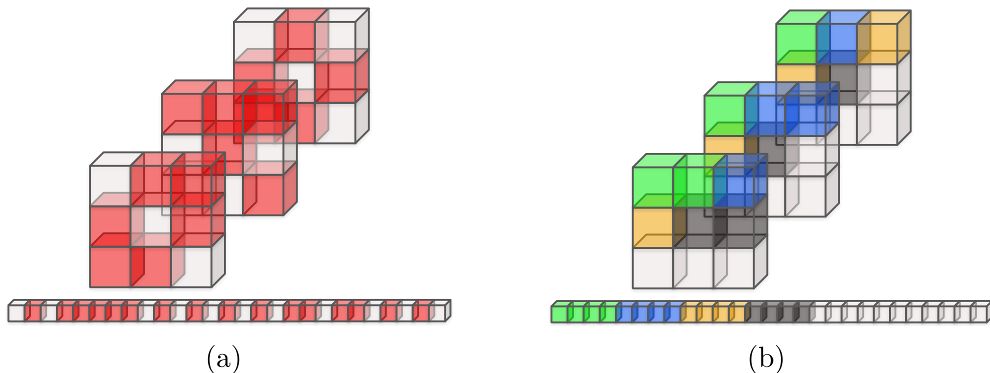


Figure 4.10: Weight pruning (a) vs. block pruning (b). Colored weights denotes zero-values.

is no direct control on the position of the zeros, which is ruled by accuracy. The EAST technique is based on the assumption that a weight pruning that is aware of the encoding scheme could make better use of sparsity. A pictorial view of this principle is given in Figure 4.10, which illustrates how multi-dimensional tensors are transformed into a 1-D array that can be processed by standard general-purpose cores. Figure (a) shows a standard weight pruning, while Figure (b) is for a block pruning with block shape $bs \in \mathbb{R}^{1\times4}$. In both cases, the picture refers to a *channel-last* layout organization (the same scheme used by the inference library adopted in this work). The colored items represent the pruned weights. While for the standard method the pruned weights are often placed far away as the selection is just accuracy-driven (smaller weights pruned first), with a block pruning the proximity of the pruned weights is forced by the size of the group itself. This brings to clear advantages. Indeed, even if both cases show the same sparsity (59%), block pruning gets 55% higher compression ratio (using LZ4, see Section 4.4.1). The savings get larger when considering tensors of higher size.

It is thereby intuitive that the block shape serves as a control knob to reach the best trade-off between accuracy, sparsity, and compression rate. When the available memory is extremely small, blocks of larger size may help to achieve higher compression with lower sparsity, hence preserving accuracy more. With a too-small group shape, e.g. $bs \in \mathbb{R}^{1\times4}$ as for standard weight pruning, the amount of sparsity needed by the encoding algorithm to meet the memory constraint would be too large, with negative effects on the accuracy. The EAST strategy implements a memory-driven adaptive block sizing during the sparse training procedure. As EAST groups the weights just on the second dimension of the *channel-last* layout, from hereafter we consider the *block size* as the m size of the generic block shape $bs \in \mathbb{R}^{n\times m}$ (in this work n is always fixed to 1).

Sparse Training. State-of-the-art sparse training strategy adopted different strategies to manage the sparsity during the optimization loop. As described in

Chapter 2, one of the most common approaches to gradually increase the sparsity during training is the polynomial decay scheduler proposed in [239]. However, EAST is storage-driven, hence needs to explore different scheduling strategies to adapt the block size and the sparsity to meet the storage constraint. In EAST, both sparsity and block size are gradually increased during the training loop until the storage constraint is met. In the beginning, the sparsity is low and the block size is set to one, hence EAST behaves like a standard weight pruning. If the storage constraint is not satisfied, sparsity and block size are updated following a pre-defined schedule. The sparse training iterates for a new bunch of epochs and if the storage memory constraint is still not met, sparsity and group size are newly updated. The larger the group size, the faster the memory reduction. Therefore, block pruning helps to converge faster attaining the target memory with a lower sparsity. The block selection is driven by the ℓ_2-norm: blocks with lower norm are removed first. However, they can be restored later during the training steps that follow, in fact, these weights can still receive gradient updates during following training epochs. Once the target is met, the sparsity and block size updates are stopped, the pruned weights are frozen, and the training iterates for the last set of epochs adjusting the remaining weights in order to maximize accuracy.

Hyperparamters. Block pruning is applied at the end of each epoch, namely after a complete iteration over the entire training set. The initial target sparsity is fixed at 30% with an increased step of 1% every epoch; the step is halved at epochs 20 and 50. The initial block size is set to one; starting from epoch 20, it increases with a step of 1 every 10 epochs.

Quantization & Encoding

After the sparse training, the 32-bit floating-point ConvNet is quantized to 8-bit. The effect of the quantization is (*i*) to reduce the memory footprint ensuring marginal accuracy loss, (*ii*) to increase the frequency of repeated weights, (*iii*) to accelerate the inference time. We opted for a binary-point quantization scheme that is fully compliant with the inference library used for on-board deployment (CMSIS-NN [112]), therefore tailored for the target MCU (the Cortex-M by ARM).

As the very last stage, the quantized model is compressed. EAST can operate different encoding algorithms, but we found the LZ4 algorithm is a good choice for resource-constrained MCUs due to its lightweight routine that ensures high decoding speed. On-board measurements validated this qualitative analysis. The implemented compression strategy is layer-wise, namely, layers are compressed as separate blocks. This solution allows more efficient management of the available SRAM as it avoids one-shot full model decoding. In fact, layers are processed in sequence during inference, therefore each layer block can be decoded independently and temporarily stored in the SRAM using time-shared buffers.

	FP32	Q8	Q8+LZ4
Top-1	91.10%	91.01%	91.01%
Memory	558kB	140kB	140kB

Table 4.8: Top-1 accuracy on CIFAR-10 and weight memory of the dense ResNet-9 after 32-bit floating-point training (FP32), after quantization (Q8), and after LZ4 compression (Q8+LZ4).

4.4.2 Experimental Results

Benchmarks, Datasets, and Training

We used as benchmark a 9-layer ResNet [193] (ResNet-9) for image classification on the CIFAR-10 dataset. ResNet-9 currently holds the first position in the DawnBench Competition [25]. In our implementation, we removed 75% of the filter from each convolutional layer. As it is already optimized for fast training and inference, this ConvNet represents a challenging test-case to assess the efficiency of different compression pipelines.

The dataset is split in training (45K images), validation (5K), and test (10K) set. The model with the highest accuracy on the validation set is selected for evaluation. For data augmentation, we applied padding with random crop, random horizontal flip, and cutout. The same setting is used for both dense and sparse training. The training is driven by SGD for 200 epochs with batch-size 128. The learning rate follows a cosine annealing schedule with an initial value of 0.1. All the experiments have been run in Pytorch 1.2.

For what concerns quantization, the fixed-point position is determined by a heuristic that minimizes the mean squared error between the floating-point and the 8-bit values. For the intermediate activations, the statistics have been collected on a sub-set of the validation set (size 100 samples).

Table 4.8 reports the top-1 accuracy on the test set and the memory size of the network. The reported values refer to a standard training (i.e. EAST off). Results confirm the efficiency of quantization (column *Q8*) that gets 4× memory reduction with negligible accuracy losses (0.09%) w.r.t. the floating-point ConvNet (column *FP32*). Applying the LZ4 compression to the quantized model does not show significant savings: just a few bytes of memory reduction (column *Q8+LZ4*).

EAST opens the deep memory space

Table 4.9 reports the comparison between a standard sparse training via weight pruning (WP) and the proposed flow built upon EAST. The two are compared for different target memories (M_t), which indicate the size of non-volatile memory (FLASH) where the model parameters are stored. The WP is trained using the

M_t	**CR**	$S_{\mathbf{WP}}$	$S_{\mathbf{EAST}}$	$A_{\mathbf{WP}}$	$A_{\mathbf{EAST}}$	ΔA
112	5.0×	58.5%	49.5%	89.80%	89.46%	-0.34%
80	7.0×	76.0%	60.5%	88.67%	88.61%	-0.06%
48	11.6×	89.5%	74.8%	87.51%	87.44%	-0.07%
40	14.0×	92.0%	79.0%	86.80%	86.82%	**0.02**%
32	17.4×	94.0%	83.3%	85.30%	86.11%	**0.81**%
24	23.3×	96.0%	87.8%	82.33%	83.65%	**1.32**%
20	27.9×	96.8%	90.0%	79.63%	81.11%	**1.48**%
16	34.9×	97.5%	91.8%	74.16%	78.45%	**4.29**%
12	46.5×	98.3%	94.0%	55.59%	64.32%	**8.73**%

Table 4.9: Comparison between state-of-the-art weight pruning (WP) and EAST in terms of Top-1 Accuracy (A) and Sparsity (S). The first column M_t indicates the set of FLASH memory constraints used to compress the ConvNets from the original dense version. The value of the final compression rate (CR) needed to meet the target is reported in the second column.

same sparsity schedule of EAST (see Section 4.4.1). For each M_{t}, the table collects the compression ratio (CR) achieved after quantization and encoding, the sparsity reached after training (columns S_{WP} and S_{EAST}), the top-1 accuracy measured on the test-set (A_{WP} and A_{EAST}) and the relative accuracy distance (ΔA) between EAST and WP. As demonstrated by previous works, when the storage constraint is met with low sparsity, weight pruning guarantees marginal accuracy losses. For instance, at M_t = 112kB the accuracy loss is only 1.21% lower than the dense 8-bit ConvNet (89.80% vs. 91.01%). In this region of memory, EAST reaches similar accuracy levels than weight pruning, 0.34% lower in the worst case (M_t = 112kB). However, in the deep memory space ($M_{\text{t}} \leq$ 40kB) weight pruning starts experiencing dramatic accuracy degradation. The reason is that very high sparsity ($>$ 90%) is needed to reach the desired storage constraint, therefore the model loses its expressive power as only a few weights remain up. In this region EAST outperforms WP; the encoding-aware pruning enables better control of the sparsity indeed ($S_{\text{EAST}} < S_{\text{WP}}$), preserving the same amount of information within the same amount of memory. On the extreme corner, M_{t} = 12kB, EAST is 8.73% more accurate than WP due to a lower sparsity (94% vs. 98.3%). To emphasize the role of EAST, one can consider that with the same amount of sparsity (e.g. 94%) the model optimized with EAST is 2.7× smaller (row 12kB vs. 32kB).

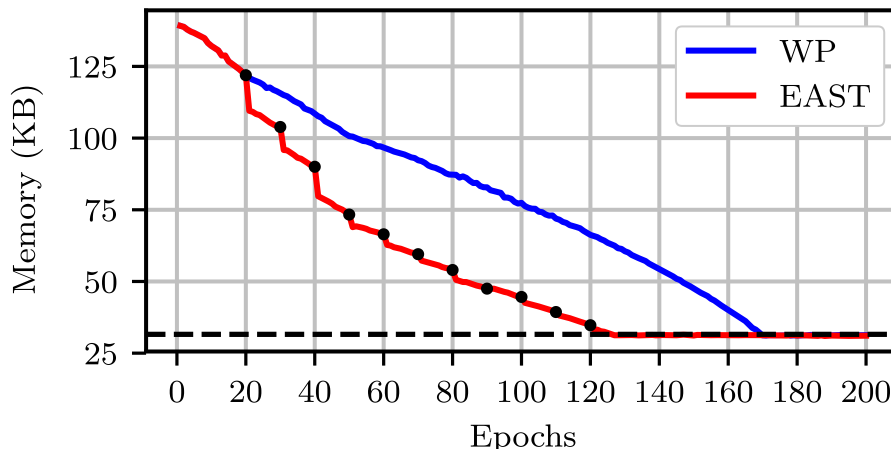


Figure 4.11: Comparison between the speeds of EAST and weight-pruning to meet the target. The line plot shows the Memory trend according to the training epochs for weight pruning (blue line) and EAST (red line). The memory target is fixed at M_t = 32kB (dashed line). Each black dot marker indicates one increment of the block size.

EAST accelerates the memory compression

Figure 4.11 shows the evolution of the memory footprint during the training epochs for both WP (blue line) and EAST (red line) under the same storage constraint M_t = 32kB. During the first 20 epochs, when the block size is one (as set by the training schedule, see Section 4.4.1), EAST follows the same trend of WP. Every time the group size gets increased (events indicated with black dots), the memory compression accelerates quickly. As a result, EAST reaches the target memory (indicated with the dashed black line) 43 epochs sooner than WP. These findings suggest that the block size works as an effective knob to boost the compression rate without seeking additional sparsity.

Efficient deployment of sparse ConvNets

We validated the optimization flow on a STM32 NUCLEO-F412ZG [153] board powered with an Arm Cortex-M4 core running at 100MHz, 1MB of FLASH memory, and 512kB of SRAM. As the inference engine, we adopted the CMSIS-NN library. The original dense ConvNet takes 28kB of SRAM to store intermediate activations and classifies a single image in 28492ms. The sparse ConvNets needs 884B of flash for the LZ4 routine, which thereby has a negligible impact on the compression rates achieved. Furthermore, an additional SRAM buffer of 28kBkB is needed to store the decompressed weights. Since this buffer is time-shared among different layers, its size is given by the biggest layer. However, the buffer can be dynamically allocated just before the execution of the ConvNet.

The total execution time is function of the storage memory constraint M_t: the larger the M_t, the longer the decompression stage. For ResNet-9 generated with EAST, the execution time ranges from 482ms at M_t = 12kB to 497ms at M_t = 112kB. At the lowest memory, the decompression only accounts for 6ms; in all cases, the network layers execute faster than the dense counterpart as the weights resides in the SRAM instead of flash.

4.4.3 Conclusions

This work opens new paths towards the optimization of ConvNets in memory-bounded cores. EAST is particularly suited for the deep memory space, where it outperforms state-of-the-art sparse training. Nevertheless, further investigation is needed to bridge the accuracy gap with dense nets at extreme constraints. First, we plan to consider other proxies than the ℓ_2-norm to drive the block selection. Second, group size and sparsity follow relative straightforward scheduling during the training; in order to achieve better trade-offs between sparsity, block size, and the position of the pruned groups, future works will explore the adoption of smarter hyper-parameter tuning techniques (e.g. Bayesian optimization or reinforcement learning) that might help EAST to reach global optima in the sparsity-memory-accuracy space.

Chapter 5

Latency-Quality Scalable ConvNets

5.1 Scalable ConvNets and Their Knobs

Recently, the design of ConvNets did not focus on the search for new topologies to improve prediction accuracy, but rather it was mostly concerned with the scalability of the existing well-known architectures. In particular, there is a set of ConvNets used by the designer patterns to rescale and resize to improve the accuracy vs. efficiency trade-off (e.g., VGG [180], MobileNets [85], ResNets [76]).

5.1.1 Static Scalability

Scaling up the size of the models is a widely used strategy to improve accuracy. However the accuracy is not the only agent, in fact, the model efficiency is crucial. For this reason, the scaling process is often not fully accomplished by the various works, as there are many ways to do it focusing on different aspects. There are three main knobs used to scale the ConvNet architectures: depth, width, and resolution.

Depth. The prior and most used approach is the depth scaling, firstly explored in [77] and followed by many works [87, 185]. The intuition is simple: a deeper model has more chance to capture more complex features from the input images. Hence, increasing the number of convolutional layers would help the model to raise its prediction capability for a given task. However, a deeper ConvNet is also more complex to train for vanishing gradient [228]. Despite skip connections [77] and batch normalization [96] can alleviate the issue on more complex training, the correlation between depth and accuracy gain rapidly slows with a high number of layers.

Width. Another very popular approach is the width scaling [228], where standard practice is to use a *width multiplier* to scale the number of channels inside each convolutional layer. This multiplicative factor refers to the ratio of channels

that respect the baseline model. This approach is commonly used for small size models [85, 172, 188]. It is well known that wider networks, i.e. with a high number of channels, tend to have a major ability to capture fine-grained features from samples, and faster training convergence [228]. However, there is a boundary over which wider networks become less capable to extract higher-level features, rapidly saturating their accuracy gain due to extremely large width.

Resolution. Then, a more recent approach is resolution scaling, which modulates the size of the input images. Samples from ILSVRC-2012 (Imagenet) benchmark [110] from original 224×224 were scaled up to 229×229 in [186] and to 331×331 to increase the prediction accuracy. Furtherly, a more recent work reached the resolution of 480×480 to achieve the state-of-the-art accuracy of Imagenet [90]. For object detection task the resolution scaling reaches higher resolutions, up to 600×600 [78, 125]. Similar to the previous case of width scaling, extremely rising up much the resolution brings a slowing of the relative accuracy gain.

5.1.2 Dynamic Scalability

However, all these strategies are very effective but they focus just on the static scaling of ConvNets, which means that a model after being trained and deployed on a device, cannot be re-scaled anymore. To overcome this limitation, more recent approaches tried to explore *dynamic* strategies to scale ConvNets with the aim to trade accuracy for speed at run time, optimizing the average resource usage. This new branch of scalable architectures can enable many vision applications, which may benefit to have multiple solution points in the latency vs. accuracy space. One example is always-on systems that require continuous monitoring of input samples, but at different priority levels: they need to employ less effort to quickly recognize some suspicious event, and more effort with slower processing to classify the type of event. Another example, in a more deterministic scenario, is to adapt the inference task to different latency constraints to meet the timing budget imposed at the system/application level for balancing the workload. This new class of scalable ConvNets are defined as *Run-time Scalable ConvNets* [201, 190, 223]. Also for the *dynamic* ConvNets, the set of scaling knobs is the same as the *static* case, but with the different aim to scale in the design space yet using a unique model.

Depth. Dynamically scaling the network depth means changing the number of layers traversed by the forward pass. This can be driven by the complexity of the current input pattern as a knob, or more pragmatically by a system-defined latency constraint. One common and simple strategy is the involvement of gating blocks, like in [205], which are modules that are embedded and trained in the ConvNet graph implementing a dynamic routing strategy, eventually with the addition of an early-exit branch [241]. The underlying full-depth model is the main contribution for the storage, in addition to the extra modules for controlling the depth.

Width. The dynamic version for scaling the number of channels (proportionally to a width multiplier [172]) has been firstly proposed in Slimmable Networks [223]. The authors introduced the concept of *switchable* batch-norm enabling a reliable training procedure for dynamic width scaling. This training strategy has been furtherly extended in [221], to execute slimmable networks at arbitrary width. The authors introduced two novel techniques, namely the *the sandwich rule* and the *in-place distillation*, with the aim to enhance the training process and boosting final accuracy. To enable to run all the possible configurations, however, these solutions need to store the full dense model on the device, despite it utilizes only a sub-set of the parameters at a time.

Resolution. Extending the resolution knob to a dynamic level is still an open issue. However very recent works propose interesting approaches to adapt ConvNet at different input sizes switching the sub-network to use. A general method to train a ConvNet able to switch resolution at run-time has been proposed in [206], enabling to switch running speed to meet different latency constraints. The ConvNet parameters are shared across the sub-networks, but they are still kept separate between batch normalization layers. A more advanced approach has been proposed in [215], where the authors proposed a width-resolution mutual learning strategy to dynamically adapt the ConvNet to different accuracy-efficiency trade-offs. At last, the dynamic scalability can be addressed also reusing a single set of quantization terms (i.e., the same set of nonzero bits in values), rather than sharing the weight values, as recently shown in [230].

5.2 Motivation

Despite the relative ease of implementation and the ability to switch settings at run-time during the inference stage, operating on the topology of the model can only offer a coarse-grained control of latency and accuracy. Moreover, it does not alleviate the pressure of the storage memory. Indeed, these strategies require storing the full model configuration, namely, the one at the maximum width, depth, or resolution. This may require more than a few MBs, which might not meet the stringent constraint of non-volatile memory (FLASH) of the edge devices. This calls for the accomplishment of a finer control knob able to modulate both the latency and the ConvNet footprint, and the sparsity is a good candidate. As reported in Chapter 2, sparse pruning has been proved to be more effective than structured pruning, as it guarantees lower accuracy losses at the same compression rates. The memory footprint of the sparse models can be effectively reduced with proper lossless encoding schema [67].

However, the use of sparsity as a dynamic knob is not trivial. The authors of [213] proposed a learning strategy to train a single ConvNet model able to work

at different sparsity levels. The reported results are positive, but they limit the analysis on Recurrent Neural Networks for Automatic Speech Recognition (ASR) [213], known to be highly over-parametrized models and hence more resilient to pruning [150]. We experimentally tested that this technique is quite frail when applied to lightweights ConvNets for computer vision tasks, where it brings substantial accuracy losses. Moreover, the authors do not deep into the real deployment and edge processing of these dynamic sparse ConvNets, which cannot be underestimated. In fact, this dynamic sparse training needs to store on the FLASH memory both the unique set of weights and a set of binary masks, one for each sparsity level. This can be a severe impediment to practical implementations on resource-constrained devices, where the non-volatile memory is often less than a few MBs.

However, deploying a dynamic sparsity model on general-purpose, low-power cores may guarantee latency-proportional processing. This issue, never discussed before, in addition to the limited results presented in [213] motivates our work: *Run-time Scalable ConvNets via Nested Sparsity.*

5.3 Run-time Scalable ConvNets via Nested Sparsity

This work contributes to the state-of-art with a novel training and compression pipeline for building *Nested Sparse ConvNets*. A Nested Sparse ConvNet consists of N nested weight-sets with programmable gradual sparsity, trained and compressed to facilitate the deployment and maximize the scaling efficiency. Specifically, we introduce:

- a novel gradient masking technique to route the learning signals between the nested sparse weight-sets, achieving better quality than existing dynamic pruning methods;
- a new sparse matrix compression format with dedicated compute kernels that fruitfully exploit the nested structure of the weight-sets, avoiding decompression stages that might impede the dynamic scaling mechanism.

To validate our proposal, we collected an extensive set of results using as benchmarks *ResNet9* [76] and two instances of *MobileNet* (V1 and V2) [85, 172] for two tiny vision tasks, i.e., image classification and object detection, ported onto a commercial embedded system powered with an ARM Cortex-M7 MCU [1]. With such use-cases, the Nested Sparse ConvNets achieved an accuracy comparable to that of independently trained sparse models and outperformed state-of-the-art scalable ConvNets obtained via dynamic sparsity [213] and layer width scaling [223], thus proving Pareto efficiency in the accuracy vs. latency space.

5.3.1 Building Nested Sparse ConvNets

Training

Training a Nested Sparse ConvNet means learning one weight-set θ and N binary masks. Each i-th pruning mask M^{s_i} serves a given sparsity level s_i and all the masks are nested, that is, more sparse masks are encapsulated in less sparse masks. Therefore, for a pre-defined set of sparsity levels $S = \{s_1, s_2, ..., s_N\}$, there is a set $M = \{M^{(s_1)}, M^{(s_2)}, ..., M^{(s_N)}\}$ that defines the N sparse weigh-sets $\theta^{(s_i)} = \theta \circ M^{(s_i)}$[2] nested in the whole weight-set Θ. In other words, θ can be seen as the weight-set of a super-network containing a set of sub-networks with multiple sparse nested weight-sets $\theta^{(s_i)}$.

In the adopted training strategy, the sub-networks are sequentially evaluated with an increasing order of sparsity, from low to high, with the weight-set θ and

[1] https://www.st.com/en/evaluation-tools/nucleo-f767zi.html

[2] $\circ$ indicates the element-wise product between two matrices, i.e., the Hadamard product.

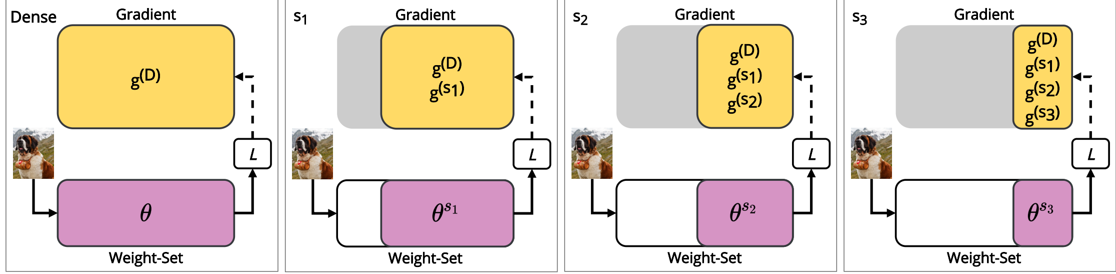


Figure 5.1: Training step: full weight-set (θ) and the sub-nets (θ^{s_i}) sorted with an increasing order of sparsity from left to right (i.e $s_1 < s_2 < s_3$).

the masks M updated only at the end of each training step. It turns out that the weights are ranked (by magnitude) just once, and all the sub-networks can share the same ordered list of weights during the pruning stage. This ensures that the N masks $M^{(s_i)}$ are nested:

$$s_1 < s_2 < \ldots < s_N \;\Rightarrow\; \theta^{(s_1)} \supset \theta^{(s_2)} \supset \ldots \supset \theta^{(s_N)} \tag{5.1}$$

The concurrent training of multiple sparse sub-networks within a single model raises several issues, among which the most critical concerns how to collect and compose the contributions coming from (and directed to) different sub-networks. The authors of [213] applied a simple average, without considering that sub-networks with different sparsity share bunches of non-pruned weights, and that the learning of such shared weights must be balanced in a proper manner. For instance, a less sparse sub-net may require a smother update compared to that of a sub-net with a higher sparsity. To accommodate this requirement, we thereby introduce a novel technique, named *gradient masking*, to guarantee a more stable and effective routing of the learning signals among the nested sub-networks. The gradient masking is integrated as part of a training loop built upon the basic structure of classical pruning and is enhanced with in-place knowledge distillation (KD). More technical details are given in the next sub-sections.

Gradient Masking A pictorial representation of one training step is reported in Figure 5.1, which depicts the dynamics of the *gradient masking*. Each frame shows a consecutive evaluation of a different sub-network, consisting of forward (solid line) and backward (dashed line) passes. L indicates the training loss. The first frame on the left (labeled as Dense) is for the full weight-set θ (i.e., $s = 0\%$), while the following ones refer to sub-networks $\theta^{(s_1)}$, $\theta^{(s_2)}$, and $\theta^{(s_3)}$, with $s_1 < s_2 < s_3$; the number of pruned weights increases from the left to the right. With gradient masking, the weights pruned within a given $\theta^{(s_i)}$ no longer contribute to the next stages, neither to the forward pass nor to the backward propagation; this is graphically depicted in Figure 5.1 with the shadowed gray regions. For instance,

the gradient computation from the sub-network with sparsity s_2 ($g^{(s_2)}$) does not interfere with the gradients previously computed for the less sparse sub-networks. The effect is twofold: first, to allow the less sparse (and possibly more accurate) sub-networks to influence the weights of the more sparse and weaker ones; second, to shield more sparse (and hence less accurate) sub-networks, preventing abrupt changes in the learning curve. This flow improves both the quality and stability of the training.

The gradient masking can be described more formally as follows. During the forward pass of each input x with label y, the local gradient for the sub-network with sparsity s_i is computed according to the SGD algorithm as:

$$\hat{g}^{(s_i)} = \frac{1}{b} \cdot \nabla\theta^{(s_i)} \cdot \sum_j L(f(x^{(j)}, \theta \circ M^{(s_i)}), y^{(j)}) \tag{5.2}$$

where $L(\cdot,\cdot)$ is loss function, $f(\cdot,\cdot)$ the forward evaluation of the network, and b the number of samples in the mini-batch. The local gradient $\hat{g}^{(s_i)}$ is then merged to the other gradient contributions through the masking operation:

$$\hat{G} = \sum_s M^{(s_i)} \circ \hat{g}^{(s_i)} \tag{5.3}$$

The same pruning mask $M^{(s_i)}$ is used to remove the contribution of the pruned set of parameters from the forward pass in Equation 5.2 and from the computation of the global gradient $\hat{G}$ in Equation 5.3, which is then used to update the full weight-set θ.

Training Loop The pseudo-code of the proposed Nested training loop is reported in Algorithm 5. It follows the basic structure of a classic block pruning procedure [72, 239] where a model is gradually pruned with a granularity of a $m\times n$ block until a target sparsity level is reached.

The procedure takes as main inputs the set of sparsity levels S, and the block shape, and it returns the weight-set θ and the set of masks M^s containing the N nested mask, one for each sparsity $s \in S$. The training loop alternates dense training stages (line 3-5) with the pruning stages (line 6-13). For each epoch e, forward and backward passes are first performed for the dense model (lines 3-5), then the weight-set θ is updated (line 14) using the gradient value, eventually after the pruning stage. During the pruning stage (line 6-13), for each sparsity level (line 7), a mask is generated through the `getMask` function and then applied to the weight-set θ (lines 8-9); the `getMask` computes the binary mask based on a L^2 magnitude policy using a given block shape. The forward and backward passes are run to compute the local gradient from the current sparse sub-network (lines 10-11), then the local gradient is masked and merged with the previous gradient contributions (line 12). To notice that the predictions of the dense model

Algorithm 5: Nested Sparse Training.

1 **Input:** epochs, S, *block_shape*
2 **for** *e in epochs* **do**
3 $\hat{G} = 0$
4 soft_labels = `forward`(θ)
5 $\hat{G}$ += `backward`(θ)
6 **if** *pruneEpoch(e)* **then**
7 **for** *s in S* **do**
8 M^s = `getMask`($\theta, s, block_shape$)
9 $\theta^s = \theta \circ M^s$
10 `forward`(θ^s, soft_labels)
11 $\hat{g}^s$ = `backward`(θ^s)
12 $\hat{G}$ += $M^s \circ \hat{g}^s$ // Gradient Masking
13 **end**
14 θ = update(θ, $\hat{G}$)
15 **end**
16 M = { `getMask`($\theta, s, block_shape$) for s $\in S$ }
17 **Output:** θ, M

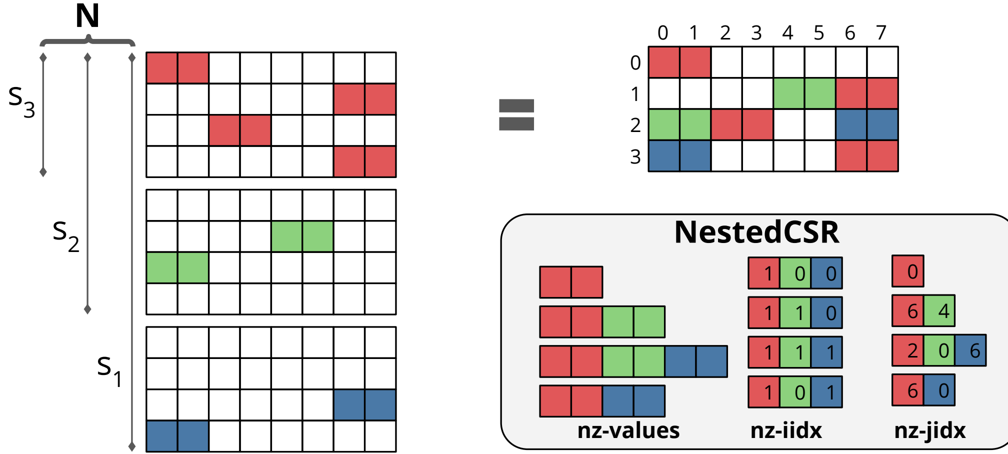


Figure 5.2: Example of the proposed storage format NestedCSR applied to a 1×2 block sparse matrix that can work in three sparsity levels $S = \{s_1, s_2, s_3\}$.

are used as soft labels, as a form of in-place distillation [221]. At last, the final weight-set and the list of nested masks are returned as the main outcome (lines 16-17).

Compression

Figure 5.2 illustrates an example of the proposed sparse matrix compression format, named NestedCSR, for a nested model trained with three generic sparsity

levels $s_1 < s_2 < s_3$ and using a block shape $bs \in \mathbb{R}^{1\times2}$. It is worth emphasizing the compression format is general and can be used with any number of sparsity levels or block sizes. At the lower sparsity (s_1), the matrix comprises the red, green, and blue non-zero blocks; at the medium sparsity (s_2), the red and green blocks; at high sparsity (s_3) the red blocks. As shown in the figure, the three configurations can be seen as a composition of three disjoint sparse matrices, and this is exactly the property exploited by NestedCSR. Each sparse sub-set is compressed using a block CSR format [225]: the *nz-values* array stores the values of the non-zero blocks in row-major order, the *nz-iidx* the number of non-zero blocks on each row, and the *nz-jidx* the column position of each non-zero blocks. The three arrays of each sparse sub-set are concatenated row-wise, from the most sparse to the least sparse (from red to blue in the example of Figure 5.2).

The footprint of a block-sparse matrix W of dimensions $R \times C$ encoded with NestedCSR depends on the size of the three arrays, defined as follows:

$$|nz\text{-}values| = (1-s_{min})\cdot|W|, \tag{5.4}$$

$$|nz\text{-}iidx| = N\cdot R, \tag{5.5}$$

$$|nz\text{-}jidx| = (1-s_{min})\cdot\frac{|W|}{n\cdot m}; \tag{5.6}$$

where N is number of sparsity levels, or nested configurations, n and m the dimensions of the non-zero block. As the main advantage of such a format, the amount of storage memory is weakly affected by the number of nested configurations. In fact, the size of *nz-iidx* is usually negligible compared to those of the other two arrays, and hence the overall footprint is set by the least sparsity adopted.

To accelerate the processing of a nested and compressed sparse layer on a general-purpose core, we implemented a custom compute kernel that follows the organization of the NestedCSR format[3]. Specifically, the kernel iterates over each row of the N sparse sub-sets selecting only those needed by the actual sparsity value, then it performs a sparse matrix multiplication [225].

5.3.2 Experimental Results

Experimental Set-up

Tasks, Datasets, and ConvNets. The proposed method was evaluated on image classification (IC) and object detection (OD) using the following data-sets.
CIFAR-10/100 (IC) [109]: 60k 32×32 RGB images annotated with 10/100 labels and split into 45k samples for training, 5k for validation, and 10k for testing.

[3] The kernel works for convolution-as-GEMM implementation of the convolutional layers [225, 112].

PASCAL VOC (OD) [40]: 15870 RGB images picked from the 2007 and 2012 PASCAL Visual Object Classes Challenge, counting of 37813 objects annotated with 20 different labels. As suggested in [128], VOC07 and VOC12 *trainval* data were used for training, while VOC07 was dedicated to testing. We limited the number of different classes to the top-10 classes recognized by the full-scale model. The image resolution was re-scaled to 160×160 with a bi-linear interpolation; this is mandatory due to the strict memory constraints of the target MCU ($512KB$ of RAM, $2MB$ of FLASH).

As ConvNet benchmarks we opted for lightweight architectures suitable for the IoT segment: a 9-layer ResNet (ResNet9) [76] for IC on CIFAR-100; MobileNetV1[85] for IC on CIFAR-10; MobileNetV2 [172] as backbone for a Single Shot Detector (SSD) [128].

Training. The training procedure for the IC task was driven by the SGD optimizer (momentum 0.9, weight decay 0.0005) for 300 epochs with batch size 128. The learning rate followed a cosine annealing schedule starting from 0.05. The same procedure was adopted for the SSD training, except for the batch-size set to 32. Images were flipped and rotated for data augmentation on the IC task, while for OD we followed the same strategy presented in [128]. Each training experiment was run three times with different seeds, and the average accuracy is reported. To train the sparse networks, both single and nested, the block shape is taken to 1×2 unit pruning with a uniform sparsity all the layers, except the first layer following a similar scheme to [38]. After 8 warm-up epochs, the sparsity levels start to increase with a polynomial decay schedule [239]. The training algorithm was implemented within the PyTorch framework ($v1.5.1$) and accelerated with a single consumer graphic card by NVIDIA (Titan Xp).

The set of sparsity levels S used to collect the results cover three values: {70%, 80%, 90%}. How to find the optimal set S to achieve the best accuracy, latency, and storage trade-off is out of the scope of this work.

In the remaining sections we refer to *Dense* as the dense baseline network, *Single Sparse* as the model optimized for a single sparsity level [72], *Nested Sparse Networks* for our proposal, *Slimmable* as the dynamic model obtained by layers width scaling [223], and *DSNN* as the dynamic sparse model [213]. For *Slimmable*, we adopted the official implementation[4], whereas for *DSNN* we used an in-house implementation as no open-source code was available at the time of writing.

Deployment. The inference latency was measured on a NUCLEO-F767ZI powered by an ARM Cortex-M7 MCU operating at $216MHz$. The board hosts 512KB of on-chip SRAM and $2MB$ of FLASH. The CMSIS-NN library v.5.6.0 [112] was

[4]https://github.com/JiahuiYu/slimmable_networks

extended by the sparse matrix multiplication kernels described in the previous section, with a block-shaped set to 1×2 to exploit the Single Instruction Multiple Data media accelerator of the M7 core [225]. To comply with the arithmetic requirements of the CMSIS-NN library, the ConvNets under analysis were quantized to 8-bit using a layer-wise symmetric binary scaling [97]. We adopted the GNU Arm Embedded Toolchain (version 6.3.1) for cross-compilation.

Nested Training Assessment

To assess the quality and generalization properties of the proposed nested training, we analyzed the accuracy achieved over the IC tasks by ConvNet architectures of decreasing capacity, that is, reduced by a width scaling factor $w \in \{1.00, 0.75, 0.50, 0.25\}$. Such scaling should not be confused with the dynamic width scaling of [223], which is discussed later (Section 5.3.2). The results are collected in Tables 5.1 and 5.2.

Nested vs. Single Sparse ConvNets Training a network for a single sparsity level should be considered as the best case because the parameters are optimized just for that specific sparsity. Contrarily, the training of a Nested Sparse ConvNet has to concurrently optimize multiple sub-networks while trying to drive each of them towards the highest accuracy. Despite the multi-objective nature of the optimization, Nested ConvNets actually perform better than individually trained sparse models in several cases, and when they achieve inferior accuracy, the gap is rather small. Considering the nested training with in-place distillation (KD), the (worst case) accuracy drop is 0.31% for MobileNetV1 and 0.96% for ResNet9.

The gradient masking technique enables less sparse sub-networks to transfer knowledge to the more sparse ones improving accuracy. The single sparse MobileNetV1@w0.25 with s=90% shows a drastic quality drop, whereas the Nested Sparse model is 20.32% more accurate, closing the gap with the less sparse configurations. To notice that the joint training does not only benefit configurations with higher sparsity, but it also improves the least sparse ones thanks to the presence of the dense model in the training loop. For instance, Nested Sparse ResNet9@w0.75 at the lowest sparsity (s=70%) is $\approx$ 1% more accurate than the single sparse model, hence much closer to the dense model.

The in-place knowledge distillation (KD) contributes effectively pushing Nested Sparse ConvNets towards higher accuracy, with an average improvement of 0.79% for MobileNetV1 and 1.41% for ResNet9. The most significant improvements can be observed on networks with a smaller width (w=$\{0.50, 0.25\}$), where KD leads up to 2.22% of accuracy gain for MobileNetV1 and up to 2.06% for ResNet9.

Nested vs. Dynamic Sparse ConvNets While the training of DSNNs [213] has proven stable on RNNs for ASR, our results reveal it performs worse on tiny ConvNets for computer vision tasks. As the worst-case, a DSNN is 3.40%

Training	Sparsity [%]	Accuracy Top-1 [%]			
		w=1.00	**w=0.75**	**w=0.50**	**w=0.25**
Dense	0	90.08	89.35	88.32	85.31
Single Sparse	70	89.70	**88.56**	87.27	**83.32**
	80	89.02	88.13	**87.04**	73.22
	90	**88.81**	86.02	75.20	57.88
DSNN [213]	70	86.30	86.21	84.09	78.84
	80	86.42	85.96	83.69	76.10
	90	85.49	84.62	81.78	72.22
Nested Sparse ConvNets					
w/o In-place KD	70	88.43	87.87	86.57	81.07
	80	88.27	87.57	86.56	80.90
	90	87.84	86.85	85.10	77.61
w/ In-place KD	70	**89.90**	88.48	**87.55**	83.29
	80	**89.20**	**88.24**	86.95	**82.12**
	90	88.50	**87.03**	**85.86**	**78.20**

Table 5.1: MobileNetV1 - CIFAR-10. Best results for each sparsity level are highlighted in bold.

less accurate than the single sparse configuration on MobileNetV1 and 13.65% on ResNet9. Except for ResNet9@w1.00 with s=90%, Nested Sparse ConvNets outperform DSNNs, getting better for more compact networks with lower width and higher sparsity. Recalling the notation introduced in Section 5.3.1, the training of DSNNs computes the global gradient as the sum of the local gradients, i.e., $\hat{G} = \sum_s \hat{g}^{(s_i)}$; we empirically demonstrated that such strategy is a source of gradient instability. In fact, our gradient masking technique overcomes this drawback thanks to a proper and selective routing of the learning signals, which has the effect of amplifying the gain brought by the joint training strategy.

Compression Pipeline Evaluation

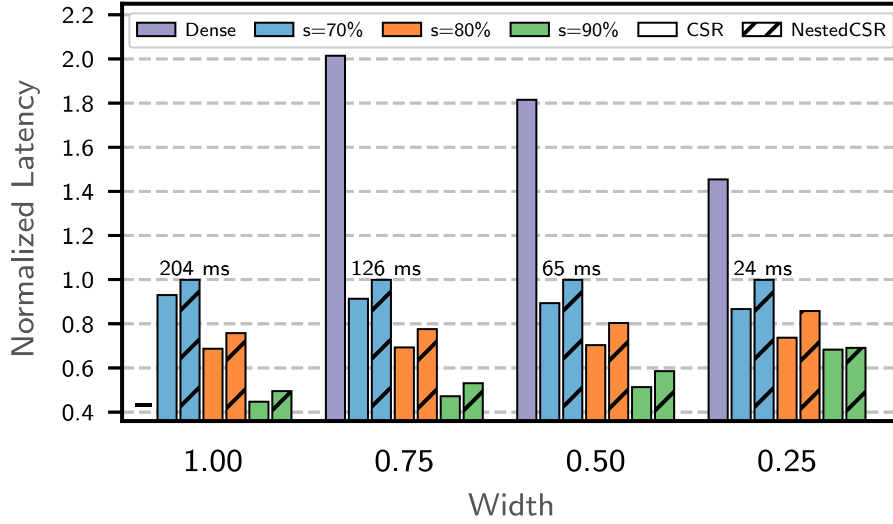
Table 5.3 reports the storage profiles for ResNet9 and MobileNetV1, showing Nested Sparse ConvNets achieve substantial savings: three sparse configurations of ResNet9@w1.00 require as low as 1016kB (54% smaller than the dense baseline), while MobileNetV1@w1.00 fits into 1464kB (53% smaller). Interestingly, a Nested Sparse ConvNet takes almost the same storage of its least sparse configuration. For instance, encoding a 70% sparse model with block CSR [225], calls for 1014KB

Training	**Sparsity [%]**	**Accuracy Top-1 [%]**			
		w=1.00	**w=0.75**	**w=0.50**	**w=0.25**
Dense	0	73.78	72.24	69.66	63.05
Single Sparse	70	72.93	71.09	68.29	**58.90**
	80	72.61	70.90	67.72	**57.40**
	90	**72.15**	**69.98**	65.04	52.15
DSNN [213]	70	72.9	70.48	63.38	45.25
	80	72.83	69.70	62.48	44.69
	90	71.62	67.56	60.15	40.92
Nested Sparse ConvNets					
w/o In-place KD	70	72.09	70.36	67.35	57.04
	80	71.72	69.97	66.91	56.53
	90	69.90	67.45	63.86	52.37
w/ In-place KD	70	**73.56**	**72.04**	**68.82**	58.70
	80	**72.94**	**71.05**	**68.38**	57.30
	90	71.19	69.59	**65.92**	**52.93**

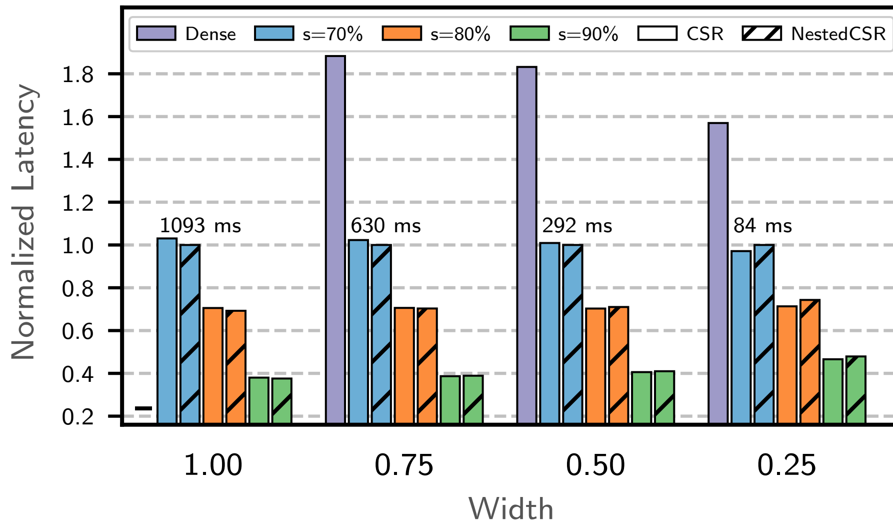
Table 5.2: ResNet9 - CIFAR-100. Best results for each sparsity level are highlighted in bold.

in the case of ResNet9@w1.00 (a mere $2kB$ less than NestedCSR) and $1458kB$ for MobileNetV1@w1.00 ($6kB$ less than NestedCSR). The models centered on the other widths follow similar trends, confirming the effectiveness of the NestedCSR format across a wide set of topology configurations.

Thanks to custom-designed compute kernels, not just memory but also latency takes advantage of the NestedCSR format. Figure 5.3 reports a comparative analysis for ResNet9 and MobileNet V1, both dense and sparse, using a classical CSR and the proposed NestedCSR. As it was expected, the sparse kernels introduce a remarkable speed-up w.r.t. the dense versions, but even more remarkable, Nested Sparse ConvNets reach comparable performance to single sparse ConvNets. For ResNet9, the multi-sparse kernels perform slightly better than single sparse kernels (1.83% on average) at high width (w=1.00 and w=0.75), and show more overhead at low width (4.04% in the worst case). For MobileNetV1, the multi-sparse kernels perform moderately worse (10.91% slower on average), and the overhead increases more notably (up to 14.08% in the worst case) for more sparse and smaller networks. Although the use of multi-sparse kernels incurs such latency penalty, it still preserves the advantage brought by dynamic sparsity and much lower storage. Multiple single sparse networks, in fact, would require storing all weight-sets on-device,



(a) MobileNetV1- CIFAR10

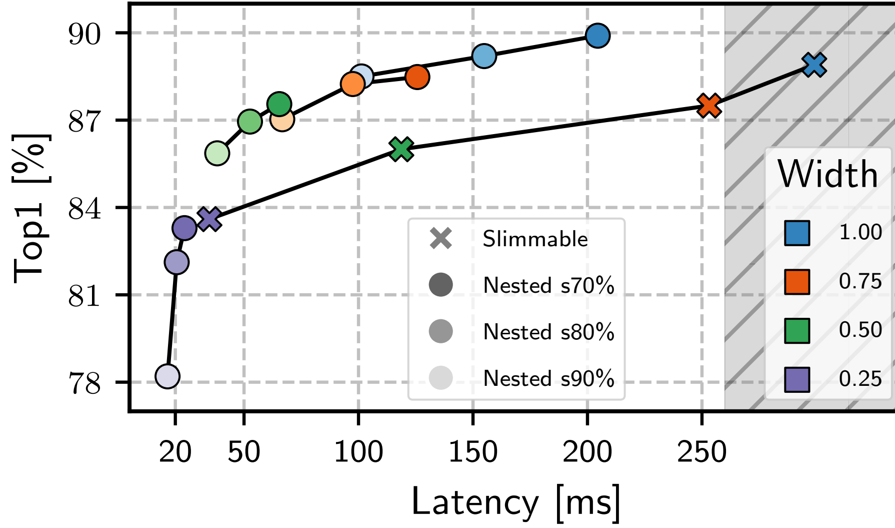


(b) ResNet9 - CIFAR100

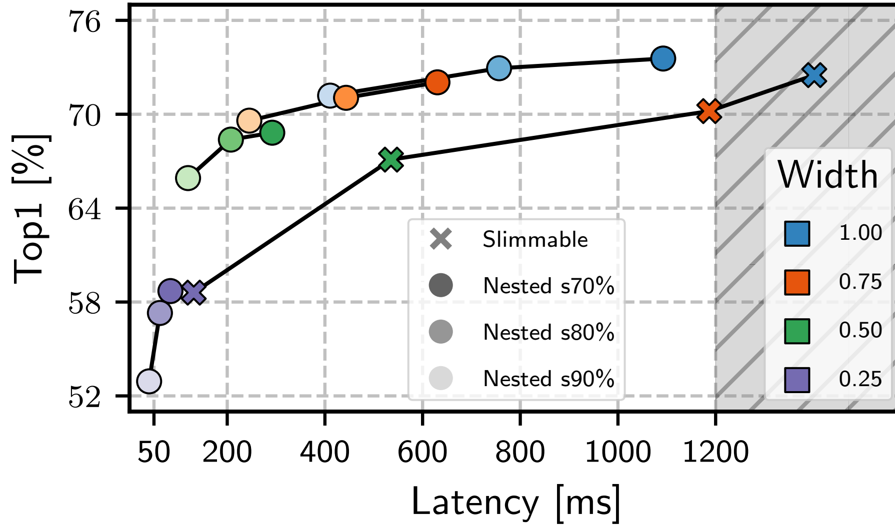
Figure 5.3: Latency values normalized for each width to the NestedCSR@s=70%. The latency of the dense model at w=1.00 is not shown as it exceeds the FLASH memory of the adopted device (2MB).

an unfeasible requirement for many tiny end-nodes.

As a side note, the latency reduction due to higher sparsity lowers down for smaller width multipliers as (i) the convolutional layers contribute less to the overall inference latency, and (ii) the lower number of channels decreases the data-reuse



(a) MobileNetV1- CIFAR10



(b) ResNet9 - CIFAR100

Figure 5.4: Latency-accuracy scaling for Slimmable ConvNets and Nested Sparse ConvNets. Grey area shows the unfeasible solution space for the adopted MCU, i.e., FLASH footprint $> 2MB$.

opportunities exploited by the sparse computational kernels. Moreover, the performance of ResNet9 and MobileNetV1 differ due to the topology of the two architectures. In MobileNetV1, there are many convolutional layers, but only the point-wise 1×1 are sparsified. Whereas, in ResNet9, there are fewer convolutional layers, but they are all sparse, and they have a higher number of channels with larger kernels (3×3). Such architectural differences lead to dissimilar performances of the sparse

Model	**Method**	**Sparsity** [%]	**Storage [KB]** w=1.00	w=0.75	w=0.50	w=0.25
MobileNetV1	Dense	0	3132	1774	800	208
	Single	70	1458	834	384	106
	Nested	$\{70, 80, 90\}$	1464	839	387	108
ResNet9	Dense	0	2232	1259	562	143
	Single	70	1014	575	260	68
	Nested	$\{70, 80, 90\}$	1016	576	260	68

Table 5.3: Storage footprint of ResNet9 trained on Cifar100 and MobileNetV1 trained on CIFAR10.

and multi-sparse kernels.

Latency-Quality Scaling

Figure 5.4 depicts the latency vs. accuracy trade-off achievable by Nested Sparse ConvNets. The best dynamic behavior is observed at the highest width. Looking at MobileNetV1@w1.00, an increase of sparsity from 70% to 90% has a minimal effect on accuracy (1.4%), but the speed-up is remarkable: 51% of latency reduction. ResNet9@w1.00 follows the same trend (Figure 5.4b), where a higher sparsity improves latency by 62% with a moderate effect on accuracy (2.37% loss). Reducing the model width makes the trade-off slightly worse as smaller ConvNets are less resilient to pruning. As a result, when the model gets smaller, the accuracy gap increases, and the latency speed-up gets lower. Still, for the smallest nets ($width$=0.25) an accuracy drop of 5.09% (5.77%) for ResNet9 (MobileNetV1) corresponds to a latency savings of 52% (31%).

To compare the sparsity knob with other architectural knobs, Figure 5.4 also reports the dynamic behavior for *Slimmable* networks [223]. Slimmable networks at maximum width w=1.00 do not satisfy the FLASH constraints ($\leq 2MB$), and only the three smaller but weaker configurations can be deployed on-device. Thanks to sparse encoding instead, Nested Sparse ConvNets of maximum width get still deployable. Except for the smallest width (w=0.25), Nested Sparse ConvNets at s=70% and s=80% turn out to be both more accurate and faster than the slimmable models. A Nested Sparse ConvNet presents a moderate scaling capacity compared to a slimmable model, which is intuitive as the sparsity acts as a fine-grain control knob on both accuracy and latency. Nevertheless, its low storage footprint paves the way to an attractive hybrid solution, where the width multiplier serves as an orthogonal static knob. The *Pareto analysis* of Figure 5.4 reveals that the three Nested Sparse ConvNets, i.e., width $\{0.75, 0.50, 0.25\}$, made scaling over the three sparsity levels, outperform the slimmable counterparts, originating eight Pareto

Training	Sparsity [%]	w=0.50 mAP [%]	w=0.50 Storage [kB]	w=0.50 Latency [ms]	w=0.35 mAP [%]	w=0.35 Storage [kB]	w=0.35 Latency [ms]
Dense	0	68.32	869	1549	63.42	523	998
Single	70	66.01	508	1080	60.58	329	752
	80	62.72	407	972	55.20	274	689
	90	29.40	306	862	23.06	219	625
Nested Sparse ConvNets							
w/ In-place KD	70	**68.30**	514	1225	**63.12**	334	883
	80	**66.37**		1103	**61.03**		807
	90	**60.33**		951	**55.84**		712

Table 5.4: SSD-MobileNetV2. Best results for each sparsity level are highlighted in bold.

optimal implementations which stored together still consume less storage than a slimmable model. Precisely, $904kB$ for ResNet9 and $1334kB$ for MobileNetV1, respectively 28% and 25% less than the deployable configurations of the *slimmable* models ($width \leq 0.75$).

Object Detection

To prove the generalization capability of our approach, we evaluated a Nested MobileNetV2 on a bounding-box detection task. Since we already demonstrated Nested Sparse ConvNets achieves better performance, in Table 5.4 we omitted the results for DSNNs for the sake of space. The results are for w=$\{0.50, 0.35\}$, which are the only fitting into the FLASH of our target MCU. The Nested Sparse object-detector gets more accurate than the sparse models trained as separate instances. More in detail, for the most sparse configurations (i.e., s=90%), it is 31.85% more accurate (average over the two widths), confirming the stability of the proposed training. As suggested in the previous subsection, a hybrid solution is still possible here: combining width scaling with nested sparsity enables scalability across a wide latency-accuracy range (ΔTop-1/ΔL = 12.46(%)/368(ms)) while cumulatively occupying $848kB$, which is still less than the single dense model at w=0.50.

5.3.3 Conclusions

To enable the run-time scalability on low-power IoT applications we proposed Nested Sparse ConvNets: a novel class of scalable models conceived to trade-off latency with accuracy at run time, leveraging sparsity as the dynamic knob. A

novel training procedure ensures high accuracy, while a custom compressed storage format together with custom compute kernels enable the deployment on tiny off-the-shelf devices. An extensive experimental assessment demonstrated that Nested Sparse ConvNets outperform state-of-the-art dynamic strategies while occupying a smaller storage footprint, making them a promising solution for expanding the adoption of adaptable computer vision tasks at the edge of the IoT.

Chapter 6

Conclusions and Future Directions

Deep neural networks have revolutionized our lives, bringing astonishing improvements in AI applications for the IoT domain. However, they require high computation and memory to be run, making very challenging their deployment on embedded systems with low power budgets and limited resources. Nowadays, there is a wide consensus among the research community that the development of near-sensor accelerators will be the enabling technology to sustain the rising computational needs of the IoT ecosystem. The hardware migration of deep learning algorithms from the cloud to the edge, however, is not trivial. The limited resources available of the tiny devices set a strong limit to the complexity of the neural networks. Such premise motivates this dissertation: making deep learning algorithms fit low-power IoT end-nodes. The aim of this dissertation is to explore new techniques and strategies to compress neural networks with hardware awareness. Reducing the requirements of memory and storage to optimize the processing of deep neural networks on embedded devices. The research has been focused on the software level optimization of these models, with the aim to explore new compression strategies able to take into consideration all the points of interest of an efficient edge inference.

At first, we presented a comprehensive overview of the most popular compression techniques used in literature. Ranging from pruning, quantization, and encoding. Describing the technical details of these strategies, we highlighted how they can be combined together to maximize optimization effectiveness. Then, we proposed several novel strategies to boost the compression of ConvNets enabling the real deployment on low-power MCUs. This aspect has been crucial, as our proposals aim to be mature for real AI applications for the IoT edge, which is an often lacking feature in previous literature.

In the context of statistically oriented compression, we focused on how to approximate the distribution of the weights to optimize ConvNets from the storage point of view. We proposed a compression-driven training framework able to optimize a ConvNet during the learning phase. The model is trained in a ternary

constrained space, where each layer learns a different boundary to approximate its distribution of weights. With a proper encoding scheme the compressed ConvNet reaches impressive benefits in terms of storage reduction and computation savings. We then extended the prior compressive training with a heuristic search to quantify the layer-wise significance of the ConvNet. The concept is that layers can be discriminated in terms of classification importance and in terms of memory footprint. The solution is a hybrid compression strategy that selects a subset of layers to compress, while leaves the others untouched. The resulting is compression boost due to the resilience of the less significant layers at the accuracy loss.

Concerning the hardware awareness in the model optimization, we studied how to design compression algorithms conscious of the hardware resources. Given a target device, the compression strategies need to accomplish all the relative hardware constraints, guaranteeing negligible accuracy losses. We first explored the memory vs. accuracy solution space to assess the optimality of ConvNet compression. With a novel hardware-conscious tool, namely *PaQ*, we showed that most of the solutions are centered on specific parameter settings that are challenging to run on low-power cores. Then we proposed a novel compression pipeline for memory-bounded ConvNets, called *VQ*. This technique enables the use of n-ary quantized ConvNets on general-purpose MCU, emulating the availability of n-ary instruction-set. At last, we presented *EAST*, a novel sparse training for storage-bounded ConvNets. The ability of this technique is to adapt the block size and the sparsity to maximize the compression rate of the weight encoding scheme.

Regarding the scalability of the ConvNets, we explored a new strategy capable to run-time scale a model in terms of latency vs. quality. State-of-the-art solutions for ConvNet scalability require multiple models to reach multiple configuration points, which can be challenging to deploy on tiny devices. To overcome this issue, we proposed a novel training and compression pipeline for building *Nested Sparse ConvNets*, a class of scalable networks that require one single nested set of parameters. The use of nested sparsity as a dynamic knob enables to run-time modulate accuracy and latency. A new custom matrix encoding with proper sparse kernels ensures efficient processing and run-time scaling of the nested weights sets. Experimental results demonstrated that nested sparse ConvNets outperform state-of-the-art dynamic strategies, yet occupying a smaller storage footprint, making them suitable for IoT applications on tiny-devices

The compression techniques presented in this dissertation offer useful insights for the optimization of the deep learning inference at the edge. It remains future work to explore the limits of model compression: what is the minimum storage needed to solve a given task, what is the minimum amount of memory to process a set of samples, and how much time the model employs to provide a correct prediction. These are some of the questions that future works may address. There is also the need to generalize these concepts beyond inference to training. The learning process of the model parameters is computation demanding and it requires

continuous fine-tuning to manage the perturbation of the source. There is the chance to improve the efficiency of the training process, like, for example, to learn the parameters in a high sparsity regime from the beginning, to reduce the training time and costs. Furthermore, concerning the IoT domain scenarios, there is an urgent need to design new strategies to continue the learning process at the edge. User-based customization of pretrained parameters, model knowledge readjustment to environment changing, incremental learning based on user experience, are some of the improvements that would be provided by the enabling of continuous learning on the edge devices, together with a higher guarantee of privacy.

Hardware and software co-design may be the key paradigm to open up more space for the artificial intelligence rise in the future of technology industries, but especially in everyone's lives.

List of Publications

This appendix lists all the publications produced during the Ph.D. years.

International Journals

- **Grimaldi, M.**, Peluso, V. and Calimera, A., 2019. Optimality assessment of memory-bounded convnets deployed on resource-constrained risc cores. IEEE Access, 7, pp.152599-152611.
- **Grimaldi, M.**, Tenace, V. and Calimera, A., 2019. Layer-Wise Compressive Training for Convolutional Neural Networks. Future Internet, 11(1), p.7.

Proceedings of International Conferences

- **Grimaldi, M.**, Peluso, V. and Calimera, A., 2020, March. EAST: Encoding-Aware Sparse Training for Deep Memory Compression of ConvNets. In 2nd International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE.
- Peluso, V., **Grimaldi, M.** and Calimera, A., 2019, October. Arbitrary-Precision Convolutional Neural Networks on Low-Power IoT Processors. In 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC) (pp. 142-147). IEEE.
- **Grimaldi, M.**, Pugliese, F., Tenace, V. and Calimera, A., 2018, October. A compression-driven training framework for embedded deep neural networks. In Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications (pp. 45-50). ACM.
- **Grimaldi, M.**, Mocerino, L., Cipolletta, A. and Calimera, A., 2021, October. Run-time Scalable ConvNets on Tiny Devices via Nested Sparsity. Under review.

Bibliography

[1] URL: https://os.mbed.com/platforms.

[2] Daehyun Ahn et al. "Double Viterbi: Weight encoding for high compression ratio and fast on-chip reconstruction for deep neural network". In: *International Conference on Learning Representations*. 2018.

[3] Jorge Albericio et al. "Cnvlutin: Ineffectual-neuron-free deep neural network computing". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 1–13.

[4] Hande Alemdar et al. "Ternary neural networks for resource-efficient AI applications". In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 2547–2554.

[5] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Structured pruning of deep convolutional neural networks". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017), pp. 1–18.

[6] Bowen Baker et al. "Accelerating neural architecture search using performance prediction". In: *arXiv preprint arXiv:1705.10823* (2017).

[7] Colby Banbury et al. "MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers". In: *arXiv preprint arXiv:2010.11267* (2020).

[8] Davis Blalock et al. "What is the state of neural network pruning?" In: *arXiv preprint arXiv:2003.03033* (2020).

[9] Yoonho Boo and Wonyong Sung. "Structured sparse ternary weight coding of deep neural networks for efficient hardware implementations". In: *2017 IEEE International workshop on signal processing systems (SiPS)*. IEEE. 2017, pp. 1–6.

[10] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[11] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[12] Han Cai, Ligeng Zhu, and Song Han. "Proxylessnas: Direct neural architecture search on target task and hardware". In: (2018).

[13] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. "An analysis of deep neural network models for practical applications". In: *arXiv preprint arXiv:1605.07678* (2016).

[14] Alessandro Capotondi et al. "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.5 (2020), pp. 871–875.

[15] Giovanna Castellano, Anna Maria Fanelli, and Marcello Pelillo. "An iterative pruning algorithm for feedforward neural networks". In: *IEEE transactions on Neural networks* 8.3 (1997), pp. 519–531.

[16] *Challenges in representation learning: Facial expression recognition challenge*. URL: http://www.kaggle.com/ (visited on 05/08/2019).

[17] Soravit Changpinyo, Mark Sandler, and Andrey Zhmoginov. "The power of sparsity in convolutional neural networks". In: *arXiv preprint arXiv:1702.06257* (2017).

[18] Changan Chen et al. "Constraint-aware deep neural network compression". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 400–415.

[19] Guobin Chen et al. "Learning efficient object detection models with knowledge distillation". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 742–751.

[20] Yu-Hsin Chen et al. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138.

[21] Wenlin Chen et al. "Compressing neural networks with the hashing trick". In: *International Conference on Machine Learning*. 2015, pp. 2285–2294.

[22] An-Chieh Cheng et al. "Searching toward pareto-optimal device-aware neural architectures". In: *Proceedings of the International Conference on Computer-Aided Design*. ACM. 2018, p. 136.

[23] Yu Cheng et al. "An exploration of parameter redundancy in deep networks with circulant projections". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 2857–2865.

[24] Aakanksha Chowdhery et al. "Visual Wake Words Dataset". In: *arXiv preprint arXiv:1906.05721* (2019).

[25] Cody Coleman et al. "Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark". In: *ACM SIGOPS Operating Systems Review* 53.1 (2019), pp. 14–25.

[26] Maxwell D Collins and Pushmeet Kohli. "Memory bounded deep convolutional networks". In: *arXiv preprint arXiv:1412.1442* (2014).

[27] Francesco Conti et al. "PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision". In: *Journal of Signal Processing Systems* 84.3 (2016), pp. 339–354.

[28] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations". In: *Advances in neural information processing systems*. 2015, pp. 3123–3131.

[29] Shail Dave et al. "Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights". In: *arXiv preprint arXiv:2007.00864* (2020).

[30] Lei Deng et al. "Gated XNOR Networks: Deep Neural Networks with Ternary Weights and Activations under a Unified Discretization Framework". In: *arXiv preprint arXiv:1705.09283* (2017).

[31] Lei Deng et al. "Model compression and hardware acceleration for neural networks: A comprehensive survey". In: *Proceedings of the IEEE* 108.4 (2020), pp. 485–532.

[32] Misha Denil et al. "Predicting parameters in deep learning". In: *Advances in neural information processing systems*. 2013, pp. 2148–2156.

[33] Peter Deutsch. *RFC1951: DEFLATE compressed data format specification version 1.3*. 1996.

[34] Peter Deutsch and Jean-Loup Gailly. *Zlib compressed data format specification version 3.3*. Tech. rep. RFC 1950, May, 1996.

[35] Xiaohan Ding et al. "Centripetal sgd for pruning very deep convolutional networks with complicated structure". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4943–4953.

[36] Xiaohan Ding et al. "Global sparse momentum sgd for pruning very deep neural networks". In: *arXiv preprint arXiv:1909.12778* (2019).

[37] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. "Learning to prune deep neural networks via layer-wise optimal brain surgeon". In: *arXiv preprint arXiv:1705.07565* (2017).

[38] Erich Elsen et al. "Fast sparse convnets". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 14629–14638.

[39] Dave Evans. "The internet of things: How the next evolution of the internet is changing everything". In: *CISCO white paper* 1.2011 (2011), pp. 1–11.

[40] Mark Everingham et al. "The pascal visual object classes (voc) challenge". In: *International journal of computer vision* 88.2 (2010), pp. 303–338.

[41] B. Liu F. Li B. Zhang. “Ternary Weight Networks”. In: *arXiv preprint arXiv:1605.04711* (2016).

[42] Javier Fernandez-Marques et al. “On-the-fly deterministic binary filters for memory efficient keyword spotting applications on embedded devices”. In: *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning*. 2018, pp. 13–18.

[43] Javier Fernández-Marqués et al. “BinaryCmd: Keyword spotting with deterministic binary basis”. In: *Proc. Conf. Syst. Mach. Learn.(SysML)*. 2018.

[44] Jonathan Frankle and Michael Carbin. “The lottery ticket hypothesis: Finding sparse, trainable neural networks”. In: *arXiv preprint arXiv:1803.03635* (2018).

[45] Jonathan Frankle et al. “The Lottery Ticket Hypothesis at Scale”. In: *arXiv preprint arXiv:1903.01611* (2019).

[46] Trevor Gale, Erich Elsen, and Sara Hooker. “The State of Sparsity in Deep Neural Networks”. In: *arXiv e-prints* arXiv:1902.09574 (2019). URL: https://arxiv.org/abs/1902.09574.

[47] Trevor Gale, Erich Elsen, and Sara Hooker. “The state of sparsity in deep neural networks”. In: *arXiv preprint arXiv:1902.09574* (2019).

[48] Trevor Gale et al. “Sparse GPU kernels for deep learning”. In: *arXiv preprint arXiv:2006.10901* (2020).

[49] *GAP8 Hardware Reference Manual*. URL: https://gwt-website-files.s3.amazonaws.com/gap8_datasheet.pdf (visited on 08/08/2019).

[50] Stéphane Gervais-Ducouret. “Next smart sensors generation”. In: *2011 IEEE Sensors Applications Symposium*. IEEE. 2011, pp. 193–196.

[51] *GNU Arm Embedded Toolchain*. URL: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm (visited on 05/08/2019).

[52] Andres Gomez, Francesco Conti, and Luca Benini. “Thermal image-based CNN’s for ultra-low power people recognition”. In: *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM. 2018, pp. 326–331.

[53] Ariel Gordon et al. “Morphnet: Fast & simple resource-constrained structure learning of deep networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1586–1595.

[54] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.

[55] Scott Gray, Alec Radford, and Diederik P Kingma. "Gpu kernels for block-sparse weights". In: *arXiv preprint arXiv:1711.09224* 3 (2017).

[56] M. Grimaldi, V. Peluso, and A. Calimera. "Optimality Assessment of Memory-Bounded ConvNets Deployed on Resource-Constrained RISC Cores". In: *IEEE Access* 7 (2019), pp. 152599–152611.

[57] Matteo Grimaldi, Valentino Peluso, and Andrea Calimera. "EAST: Encoding-Aware Sparse Training for Deep Memory Compression of ConvNets". In: *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2020, pp. 233–237.

[58] Matteo Grimaldi, Valerio Tenace, and Andrea Calimera. "Layer-wise compressive training for convolutional neural networks". In: *Future Internet* 11.1 (2019), p. 7.

[59] Matteo Grimaldi et al. "A compression-driven training framework for embedded deep neural networks". In: *Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications*. 2018, pp. 45–50.

[60] Luis Guerra et al. "Switchable Precision Neural Networks". In: *arXiv preprint arXiv:2002.02815* (2020).

[61] Yiwen Guo, Anbang Yao, and Yurong Chen. "Dynamic network surgery for efficient dnns". In: *arXiv preprint arXiv:1608.04493* (2016).

[62] Suyog Gupta et al. "Deep learning with limited numerical precision". In: *International Conference on Machine Learning*. 2015, pp. 1737–1746.

[63] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. "Hardware-oriented approximation of convolutional neural networks". In: *arXiv preprint arXiv:1604.03168* (2016).

[64] Philipp Gysel et al. "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks". In: *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5784–5789.

[65] David Ha, Andrew Dai, and Quoc V Le. "Hypernetworks". In: *arXiv preprint arXiv:1609.09106* (2016).

[66] Masafumi Hagiwara. "Removal of hidden units and weights for back propagation networks". In: *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*. Vol. 1. IEEE. 1993, pp. 351–354.

[67] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[68] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations, ICLR 2016*. 2016.

[69] Song Han et al. "Dsd: Dense-sparse-dense training for deep neural networks". In: *arXiv preprint arXiv:1607.04381* (2016).

[70] Song Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 243–254. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.30. URL: https://doi.org/10.1109/ISCA.2016.30.

[71] Song Han et al. "EIE: efficient inference engine on compressed deep neural network". In: *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 243–254.

[72] Song Han et al. "Learning both weights and connections for efficient neural network". In: *Advances in Neural Information Processing Systems*. 2015, pp. 1135–1143.

[73] Stephen Hanson and Lorien Pratt. "Comparing biases for minimal network construction with back-propagation". In: *Advances in neural information processing systems* 1 (1988), pp. 177–185.

[74] Soheil Hashemi et al. "Understanding the impact of precision quantization on the accuracy and energy of neural networks". In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2017, pp. 1474–1479.

[75] Babak Hassibi, David G Stork, and Gregory J Wolff. "Optimal brain surgeon and general network pruning". In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 293–299.

[76] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[77] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[78] Kaiming He et al. "Mask r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[79] Yang He et al. "Filter pruning via geometric median for deep convolutional neural networks acceleration". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4340–4349.

[80] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel pruning for accelerating very deep neural networks”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 1389–1397.

[81] Yihui He et al. “Amc: Automl for model compression and acceleration on mobile devices”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 784–800.

[82] Marti A. Hearst et al. “Support vector machines”. In: *IEEE Intelligent Systems and their applications* 13.4 (1998), pp. 18–28.

[83] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the knowledge in a neural network”. In: *arXiv preprint arXiv:1503.02531* (2015).

[84] Torsten Hoefler et al. “Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks”. In: *arXiv preprint arXiv:2102.00554* (2021).

[85] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[86] Hengyuan Hu et al. “Network trimming: A data-driven neuron pruning approach towards efficient deep architectures”. In: *arXiv preprint arXiv:1607.03250* (2016).

[87] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[88] Gao Huang et al. “Multi-Scale Dense Networks for Resource Efficient Image Classification”. In: *International Conference on Learning Representations*. 2018.

[89] Yanping Huang et al. “Gpipe: Efficient training of giant neural networks using pipeline parallelism”. In: *Advances in neural information processing systems* 32 (2019), pp. 103–112.

[90] Zehao Huang and Naiyan Wang. “Data-driven sparse structure selection for deep neural networks”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 304–320.

[91] Itay Hubara et al. “Binarized neural networks”. In: *Advances in neural information processing systems* 29 (2016), pp. 4107–4115.

[92] Itay Hubara et al. “Quantized neural networks: Training neural networks with low precision weights and activations”. In: *arXiv preprint arXiv:1609.07061* (2016).

[93] Itay Hubara et al. “Quantized neural networks: Training neural networks with low precision weights and activations”. In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898.

[94] David A Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[95] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).

[96] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[97] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[98] Benoit Jacob et al. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[99] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[100] Xiaojie Jin et al. “Training skinny deep neural networks with iterative hard thresholding methods”. In: *arXiv preprint arXiv:1607.05423* (2016).

[101] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *arXiv preprint arXiv:1704.04760* (2017).

[102] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *arXiv preprint arXiv:1704.04760* (2017).

[103] Nal Kalchbrenner et al. “Efficient neural audio synthesis”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2410–2419.

[104] Ye-Hoon Kim et al. “Nemo: Neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy”. In: *ICML 2017 AutoML Workshop*. 2017.

[105] William F Kindel, Elijah D Christensen, and Joel Zylberberg. “Using deep learning to reveal the neural code for images in primary visual cortex”. In: *arXiv preprint arXiv:1706.06208* (2017).

[106] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[107] Diederik P Kingma, Tim Salimans, and Max Welling. "Variational dropout and the local reparameterization trick". In: *arXiv preprint arXiv:1506.02557* (2015).

[108] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[109] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[110] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[111] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[112] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus". In: *arXiv preprint arXiv:1801.06601* (2018).

[113] Ya Le and Xuan Yang. "Tiny imagenet visual recognition challenge". In: *CS 231N* 7 (2015).

[114] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[115] Yann LeCun, John S Denker, and Sara A Solla. "Optimal brain damage". In: *Advances in neural information processing systems*. 1990, pp. 598–605.

[116] Dongsoo Lee et al. "Viterbi-based pruning for sparse matrix with fixed and high index compression ratio". In: *International Conference on Learning Representations*. 2018.

[117] Edward H Lee et al. "Lognet: Energy-efficient neural networks using logarithmic computation". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2017, pp. 5900–5904.

[118] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. "Snip: Single-shot network pruning based on connection sensitivity". In: *arXiv preprint arXiv:1810.02340* (2018).

[119] Hao Li et al. "Pruning filters for efficient convnets". In: *arXiv preprint arXiv:1608.08710* (2016).

[120] He Li, Kaoru Ota, and Mianxiong Dong. "Learning IoT in edge: deep learning for the internet of things with edge computing". In: *IEEE Network* 32.1 (2018), pp. 96–101.

[121] Xiang Li et al. "Selective kernel networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2019, pp. 510–519.

[122] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. "Fixed point quantization of deep convolutional networks". In: *International conference on machine learning*. 2016, pp. 2849–2858.

[123] Ji Lin et al. "MCUNet: Tiny Deep Learning on IoT Devices". In: *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 2020.

[124] Ji Lin et al. "Runtime neural pruning". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 2178–2188.

[125] Tsung-Yi Lin et al. "Feature pyramid networks for object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2117–2125.

[126] Baoyuan Liu et al. "Sparse convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 806–814.

[127] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *International Conference on Learning Representations*. 2018.

[128] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[129] Zechun Liu et al. "Metapruning: Meta learning for automatic neural network channel pruning". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 3296–3305.

[130] Zhi-Gang Liu, Paul N Whatmough, and Matthew Mattina. "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference". In: *IEEE Computer Architecture Letters* 19.1 (2020), pp. 34–37.

[131] Zhuang Liu et al. "Learning efficient convolutional networks through network slimming". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 2736–2744.

[132] Antonio Loquercio et al. "Dronet: Learning to fly by driving". In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 1088–1095.

[133] Christos Louizos, Karen Ullrich, and Max Welling. "Bayesian compression for deep learning". In: *arXiv preprint arXiv:1705.08665* (2017).

[134] Christos Louizos, Max Welling, and Diederik P Kingma. "Learning Sparse Neural Networks through L_0 Regularization". In: *International Conference on Learning Representations*. 2018.

[135] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. "Thinet: A filter level pruning method for deep neural network compression". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 5058–5066.

[136] *LZ4*. URL: https://lz4.github.io/lz4/.

[137] Andrew L Maas et al. "Learning word vectors for sentiment analysis". In: *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*. Association for Computational Linguistics. 2011, pp. 142–150.

[138] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. "Fog computing: A taxonomy, survey and future directions". In: *Internet of everything*. Springer, 2018, pp. 103–130.

[139] Huizi Mao et al. "Exploring the regularity of sparse structure in convolutional neural networks". In: *arXiv preprint arXiv:1705.08922* (2017).

[140] Manu Mathew et al. "Sparse, quantized, full frame cnn for low power embedded devices". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2017, pp. 11–19.

[141] H Brendan McMahan et al. "Communication-efficient learning of deep networks from decentralized data". In: *arXiv preprint arXiv:1602.05629* (2016).

[142] *MEMS and Sensors - STMicroelectronics*. URL: st.com/en/mems-and-sensors.html (visited on 05/08/2019).

[143] Asit Mishra and Debbie Marr. "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy". In: *arXiv preprint arXiv:1711.05852* (2017).

[144] Deepak Mittal et al. "Recovering from random pruning: On the plasticity of deep convolutional neural networks". In: *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2018, pp. 848–857.

[145] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. "Variational dropout sparsifies deep neural networks". In: *arXiv preprint arXiv:1701.05369* (2017).

[146] Pavlo Molchanov et al. "Importance estimation for neural network pruning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 11264–11272.

[147] Pavlo Molchanov et al. "Pruning convolutional neural networks for resource efficient inference". In: *arXiv preprint arXiv:1611.06440* (2016).

[148] Bert Moons et al. "An energy-efficient precision-scalable ConvNet processor in 40-nm CMOS". In: *IEEE Journal of solid-state Circuits* 52.4 (2017), pp. 903–914.

[149] Bert Moons et al. "Energy-efficient convnets through approximate computing". In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2016, pp. 1–8.

[150] Sharan Narang et al. "Exploring sparsity in recurrent neural networks". In: *arXiv preprint arXiv:1704.05119* (2017).

[151] Kirill Neklyudov et al. "Structured bayesian pruning via log-normal multiplicative noise". In: *arXiv preprint arXiv:1705.07283* (2017).

[152] *NUCLEO-F412ZG*. URL: https://www.st.com/en/evaluation-tools/nucleo-f412zg.html (visited on 08/08/2019).

[153] *NUCLEO-F412ZG*. URL: https://www.st.com/en/evaluation-tools/nucleo-f412zg.html (visited on 05/08/2019).

[154] *NUCLEO-F767ZI*. URL: https://www.st.com/en/evaluation-tools/nucleo-f767zi.html (visited on 08/08/2019).

[155] *Nucleo-F767ZI*. URL: https://www.st.com/en/evaluation-tools/nucleo-f767zi.html.

[156] G. Ottavi et al. "A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 512–517. DOI: 10.1109/ISVLSI49217.2020.000-5.

[157] Angshuman Parashar et al. "Scnn: An accelerator for compressed-sparse convolutional neural networks". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 27–40.

[158] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017.

[159] Valentino Peluso and Andrea Calimera. "Scalable-effort convnets for multilevel classification". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[160] Valentino Peluso, Matteo Grimaldi, and Andrea Calimera. "Arbitrary-Precision Convolutional Neural Networks on Low-Power IoT Processors". In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2019, pp. 142–147.

[161] Valentino Peluso et al. "Enabling energy-efficient unsupervised monocular depth estimation on armv7-based platforms". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1703–1708.

[162] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *arXiv preprint arXiv:1802.05668* (2018).

[163] Adam Polyak and Lior Wolf. "Channel-level acceleration of deep face representations". In: *IEEE Access* 3 (2015), pp. 2163–2175.

[164] Lutz Prechelt. "Early stopping-but when?" In: *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69.

[165] Jiantao Qiu et al. "Going deeper with embedded fpga platform for convolutional neural network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 26–35.

[166] Qiang Qiu, Xiuyuan Cheng, Guillermo Sapiro, et al. "Dcfnet: Deep neural network with decomposed convolutional filters". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4198–4207.

[167] Maithra Raghu et al. "On the expressive power of deep neural networks". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 2847–2854.

[168] Mohammad Rastegari et al. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[169] Frank Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological review* 65.6 (1958), p. 386.

[170] Manuele Rusci et al. "Quantized NNs as the definitive solution for inference on low-power ARM MCUs? work-in-progress". In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 2018, pp. 1–2.

[171] Tara N Sainath and Carolina Parada. "Convolutional neural networks for small-footprint keyword spotting". In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[172] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[173] Victor Sanh, Thomas Wolf, and Alexander M Rush. "Movement pruning: Adaptive sparsity by fine-tuning". In: *arXiv preprint arXiv:2005.07683* (2020).

[174] Abigail See, Minh-Thang Luong, and Christopher D Manning. "Compression of neural machine translation models via pruning". In: *arXiv preprint arXiv:1606.09274* (2016).

[175] Lorenzo Seidenari et al. "Deep artwork detection and retrieval for automatic context-aware audio guides". In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 13.3s (2017), p. 35.

[176] Hardik Sharma et al. "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press. 2018, pp. 764–775.

[177] Tao Sheng et al. "A quantization-friendly separable convolution for mobilenets". In: *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE. 2018, pp. 14–18.

[178] Weisong Shi et al. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646.

[179] Jocelyn Sietsma. "Neural net pruning-why and how". In: *Proceedings of International Conference on Neural Networks, San Diego, CA, 1988*. Vol. 1. 1988, pp. 325–333.

[180] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[181] Richard Socher et al. "Parsing natural scenes and natural language with recursive neural networks". In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. 2011, pp. 129–136.

[182] Suraj Srinivas and R Venkatesh Babu. "Data-free parameter pruning for deep neural networks". In: *arXiv preprint arXiv:1507.06149* (2015).

[183] Kenji Suzuki, Isao Horiba, and Noboru Sugie. "A simple neural network pruning algorithm with application to filter synthesis". In: *Neural processing letters* 13.1 (2001), pp. 43–53.

[184] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[185] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[186] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[187] Mingxing Tan and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6105–6114.

[188] Mingxing Tan et al. "Mnasnet: Platform-aware neural architecture search for mobile". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[189] Hokchhay Tann et al. "Hardware-Software Codesign of Accurate, Multiplier-free Deep Neural Networks". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM. 2017, p. 28.

[190] Hokchhay Tann et al. "Runtime configurable deep neural networks for energy-accuracy trade-off". In: *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2016, pp. 1–10.

[191] Lucas Theis et al. "Faster gaze prediction with dense networks and fisher pruning". In: *arXiv preprint arXiv:1801.05787* (2018).

[192] Philippe Tillet, HT Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019, pp. 10–19.

[193] *TorchSkeleton*. URL: https://github.com/wbaek/torchskeleton (visited on 05/08/2019).

[194] VW-S Tseng et al. "Deterministic binary filters for convolutional neural networks". In: International Joint Conferences on Artificial Intelligence Organization. 2018.

[195] Frederick Tung et al. "CLIP-Q: Deep network compression learning by in-parallel pruning-quantization". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7873–7882.

[196] Karen Ullrich, Edward Meeds, and Max Welling. "Soft weight-sharing for neural network compression". In: *arXiv preprint arXiv:1702.04008* (2017).

[197] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 307–3077.

[198] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. "Improving the speed of neural networks on CPUs". In: *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*. Vol. 1. Citeseer. 2011, p. 4.

[199] Aravind Vasudevan, Andrew Anderson, and David Gregg. "Parallel multi channel convolution using general matrix multiplication". In: *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*. IEEE. 2017, pp. 19–24.

[200] Stylianos I Venieris, Javier Fernandez-Marques, and Nicholas D Lane. "unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-Fly Weights Generation". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2021, pp. 165–175.

[201] Swagath Venkataramani et al. "Scalable-effort classifiers for energy-efficient machine learning". In: *Proceedings of the 52nd Annual Design Automation Conference*. 2015, pp. 1–6.

[202] Sebastian Vogel et al. "Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base". In: *Proceedings of the International Conference on Computer-Aided Design*. ACM. 2018, p. 9.

[203] Chaoqi Wang et al. "Eigendamage: Structured pruning in the kronecker-factored eigenbasis". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6566–6575.

[204] Kuan Wang et al. "Haq: Hardware-aware automated quantization with mixed precision". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 8612–8620.

[205] Xin Wang et al. "Skipnet: Learning dynamic routing in convolutional networks". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 409–424.

[206] Yikai Wang et al. "Resolution switchable networks for runtime efficient image recognition". In: *European Conference on Computer Vision*. Springer. 2020, pp. 533–549.

[207] Zhisheng Wang, Jun Lin, and Zhongfeng Wang. "Hardware-oriented compression of long short-term memory for efficient inference". In: *IEEE Signal Processing Letters* 25.7 (2018), pp. 984–988.

[208] Pete Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *arXiv preprint arXiv:1804.03209* (2018).

[209] Wei Wen et al. "Learning structured sparsity in deep neural networks". In: *arXiv preprint arXiv:1608.03665* (2016).

[210] *Why the PowerVR Series2NX NNA is the future of neural net acceleration.* URL: https://www.imgtec.com/blog/why-the-powervr-2nx-nna-is-the-future-of-neural-net-acceleration/e (visited on 04/08/2019).

[211] Bichen Wu et al. "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10734–10742.

[212] Carole-Jean Wu et al. "Machine learning at facebook: Understanding inference at the edge". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 331–344.

[213] Zhaofeng Wu et al. "Dynamic sparsity neural networks for automatic speech recognition". In: *arXiv preprint arXiv:2005.10627* (2020).

[214] Xia Xiao, Zigeng Wang, and Sanguthevar Rajasekaran. "Autoprune: Automatic network pruning by regularizing auxiliary parameters". In: *Advances in neural information processing systems* 32 (2019).

[215] Taojiannan Yang et al. "Mutualnet: Adaptive convnet via mutual learning from network width and resolution". In: *European Conference on Computer Vision*. Springer. 2020, pp. 299–315.

[216] Tien-Ju Yang et al. "Netadapt: Platform-aware neural network adaptation for mobile applications". In: *Energy* 41 (2018), p. 46.

[217] Yingzhen Yang et al. "Fsnet: Compression of deep convolutional neural networks by filter summary". In: *arXiv preprint arXiv:1902.03264* (2019).

[218] Penghang Yin et al. "Quantization and Training of Low Bit-Width Convolutional Neural Networks for Object Detection". In: *arXiv preprint arXiv:1612.06052* (2016).

[219] Zhonghui You et al. "Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks". In: *arXiv preprint arXiv:1909.08174* (2019).

[220] Dong Yu et al. "Exploiting sparseness in deep neural networks for large vocabulary speech recognition". In: *2012 IEEE International conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2012, pp. 4409–4412.

[221] Jiahui Yu and Thomas S Huang. "Universally slimmable networks and improved training techniques". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1803–1811.

[222] Jiahui Yu and Thomas S Huang. "Universally slimmable networks and improved training techniques". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1803–1811.

[223] Jiahui Yu et al. "Slimmable Neural Networks". In: *International Conference on Learning Representations*. 2018.

[224] Jiecao Yu et al. "Scalpel: Customizing dnn pruning to the underlying hardware parallelism". In: *ACM SIGARCH Computer Architecture News*. Vol. 45. 2. ACM. 2017, pp. 548–560.

[225] Jiecao Yu et al. "Scalpel: Customizing dnn pruning to the underlying hardware parallelism". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 548–560.

[226] Ruichi Yu et al. "Nisp: Pruning networks using neuron importance score propagation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9194–9203.

[227] Ming Yuan and Yi Lin. "Model selection and estimation in regression with grouped variables". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.1 (2006), pp. 49–67.

[228] Sergey Zagoruyko and Nikos Komodakis. “Wide residual networks”. In: *arXiv preprint arXiv:1605.07146* (2016).

[229] Matthew D Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[230] Sai Qian Zhang et al. “Training for multi-resolution inference using reusable quantization terms”. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021, pp. 845–860.

[231] Ying Zhang et al. “Deep mutual learning”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4320–4328.

[232] Ying Zhang et al. “Deep mutual learning”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4320–4328.

[233] Yundong Zhang et al. “Hello edge: Keyword spotting on microcontrollers”. In: *arXiv preprint arXiv:1711.07128* (2017).

[234] Hao Zhou, Jose M Alvarez, and Fatih Porikli. “Less is more: Towards compact cnns”. In: *European Conference on Computer Vision*. Springer. 2016, pp. 662–677.

[235] Hattie Zhou et al. “Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/1113d7a76ffceca1bb350bfe145467c6-Paper.pdf.

[236] Shuchang Zhou et al. “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients”. In: *arXiv preprint arXiv:1606.06160* (2016).

[237] Chenzhuo Zhu et al. “Trained ternary quantization”. In: *arXiv preprint arXiv:1612.01064* (2016).

[238] Maohua Zhu et al. “Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 359–371.

[239] Michael Zhu and Suyog Gupta. “To prune, or not to prune: exploring the efficacy of pruning for model compression”. In: *arXiv preprint arXiv:1710.01878* (2017).

[240] Wei Zhu et al. “Scale-equivariant neural networks with decomposed convolutional filters”. In: *arXiv preprint arXiv:1909.11193* (2019).

[241] Xinqi Zhu and Michael Bain. "B-CNN: branch convolutional neural network for hierarchical classification". In: *arXiv preprint arXiv:1709.09890* (2017).

[242] Tao Zhuang et al. "Neuron-level Structured Pruning using Polarization Regularizer". In: *Advances in Neural Information Processing Systems* 33 (2020).

[243] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016).

[244] Barret Zoph et al. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018, pp. 8697–8710.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.