Politecnico di Torino | ScuDo

Scuola di Dottorato ~ Doctoral School

WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Computer and Control Engineering (37^{th}cycle)

Automatic Hardware-aware Design and Optimization of Deep Learning Models

By

Matteo Risso

******

Supervisor(s):

Prof. Massimo Poncino, Supervisor

Prof. Daniele Jahier Pagliari, Co-Supervisor

Doctoral Examination Committee:

Politecnico di Torino

2025

Declaration

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

Matteo Risso
2025

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

I dedicate this thesis to my grandfathers, Piero and Renzo

Acknowledgements

Those who know me even a little are aware that writing—especially writing about myself and my feelings—is not my favourite thing. Still, a PhD comes only once in a lifetime, and this feels like the right moment to look back at the road travelled and thank the people who walked with me.

First and foremost, I wish to thank my supervisors Massimo Poncino, Daniele Jahier Pagliari, and Alessio Burrello (the latter not listed on the title page, yet a supervisor in every meaningful sense). From day one, you have guided me with patience and enthusiasm. I still remember when, during the spring of the COVID-19 pandemic, you guided me to turn the first encouraging results of my thesis into a paper; since then, three relentless years have flown by. I hope you are as proud of the outcome of this research as I am.

To everyone who orbits the Lab4 ecosystem: you are too many to name without risking injustice, but I cannot help to cite out—alphabetically—Adelaide, Fabio, Francesco, Giovanni, Luca, and Mohamed. Though I love working from the quiet of my house, knowing I would meet you was reason enough to show up at the lab (every now and then).

I am deeply grateful to my family, especially my parents, Angelo and Magda, and my grandmothers, Marta and Rosanna. You know I struggle to say these things aloud and need occasions like this one, but thank you. I love you.

Thank you, Claudia and Elio Fabrizio, for sharing the same roof with me 24/7 for more than a year now. Thank you for being there and for creating something whose whole is greater than the sum of its parts.

Finally, I dedicate this thesis to my grandfathers, Piero and Renzo. Only now do I realize that you were the first two men of science I ever met.

Abstract

Deep Learning represents one of the major technological breakthroughs of recent years. Typically, the training and inference phases of Deep Learning models, known as Deep Neural Networks (DNNs), happen on powerful cloud hardware. Nonetheless, evidence exists about the potential advantages of directly deploying DNNs on resource-constrained edge devices. These advantages range in several directions, including reduced latency, reduced energy consumption, improved privacy, and enhanced reliability.

Nonetheless, deploying DNNs on edge devices represents a non-negligible challenge due to resource constraints regarding memory, operating frequencies, and supported instructions. This thesis will present several optimization techniques that can consider the target hardware's characteristics, such as the memory footprint or the latency, to enable DNN inference at the edge. Moreover, all the novel techniques presented in this manuscript are framed as gradient-based optimization problems aimed at minimizing a combination of task-specific loss and hardware-aware cost metrics. The optimization knobs are the standard and the architectural parameters of the DNN which are learned through gradient descent. In this way, the DNN can be trained and optimized in one shot.

The first optimization technique presented will be the structured pruning approach known as PIT. This algorithm allows exploring the key hyperparameters of Temporal Convolutional Networks (TCNs), a state-of-the-art DNN typically used to process time-series data such as audio and bio-signals.

Then, we will show how to jointly explore structured pruning with mixed-precision search in Convolutional Neural Networks (CNNs). With this algorithm, it is possible to completely remove or quantize each weight channel to different precisions in each CNN layer.

Third, we will propose a novel optimization problem formulation that introduces multiple hardware-related constraints. This formulation allows us to discover DNNs in the design space that perform well on the task at hand while simultaneously satisfying multiple constraints imposed by the target hardware platform.

The last optimization tool discussed will be ODiMO, a mapping tool capable of splitting the execution of DNNs with fine-grain over multiple compute units available in System on Chips (SoCs) that accelerate different DNN layer alternatives or with operands with incompatible precision.

Finally, the manuscript will conclude with three real-world applications where DNNs are optimized using some of the techniques discussed above and deployed on edge devices. These applications include PPG-based heart rate estimation on wearables, visual-pose estimation on nano-drones, and people counting on low-resolution infrared arrays.

Contents

List of Figures **xii**

List of Tables **xvii**

1 Introduction **1**

1.1 On the Absence of Attention-based Transformers 4

1.2 Thesis Structure 5

1.3 List of Publications 7

2 Background **10**

2.1 DNNs and Gradient-based Learning 11

2.1.1 Deep Neural Networks 11

2.1.2 Gradient-based Learning 16

2.1.3 Deep Learning Frameworks 19

2.2 Optimization Techniques 20

2.2.1 Pruning 21

2.2.2 Neural Architecture Search 22

2.2.3 Quantization and Mixed-Precision Search 26

2.2.4 Knowledge Distillation 27

2.3 Hardware Targets 28

2.3.1 STM32 28

2.3.2 GAP8 29
2.3.3 Hardware with Mixed-Precision Support 30
2.3.4 Multi-accelerator System-on-Chips 31
2.3.5 HWatch Platform 33
2.3.6 MAUPITI 34

3 Structured Pruning of Temporal Convolutional Networks 36

3.1 Related Works 37
3.2 Proposed Method 39
3.2.1 Search-Space 39
3.2.2 Cost-aware Regularization 47
3.2.3 Training Procedure 49
3.3 Benchmarks 50
3.3.1 PPG-based Heart-Rate Monitoring 50
3.3.2 ECG-based Arrhythmia Detection 51
3.3.3 Keyword Spotting 52
3.4 Experimental Results 52
3.4.1 Search Space Exploration 53
3.4.2 Ablation Studies 55
3.4.3 Comparison with SotA tools 57
3.4.4 Embedded Deployment 60

4 Joint Pruning and Channel-wise Mixed-Precision Quantization 62

4.1 Related Works 63
4.1.1 Mixed-Precision Search 63
4.1.2 Joint Quantization and Pruning 64
4.2 Proposed Method 65

4.2.1 Search Space 66
4.2.2 Optimization Method 68
4.2.3 Complexity Regularizers 70
4.2.4 Training Procedure 73
4.2.5 Implementation details 74
4.3 Experimental Results 76
4.3.1 Experimental Setup 76
4.3.2 Sampling Methods Comparison 77
4.3.3 State-of-the-art Comparison 78
4.3.4 Deployment 80
4.3.5 Ablation studies 82

5 DNN Optimization with Multiple Hardware Constraints 86

5.1 Related Works 87
5.2 Proposed Method 88
5.2.1 Path-based DNAS 90
5.2.2 Mask-based DNAS 91
5.2.3 Multi-Constraint Loss Formulation 92
5.2.4 Training Procedure 96
5.3 Experimental Results 100
5.3.1 Experimental Setup 100
5.3.2 Global Constraints 101
5.3.3 Layer-wise Constraints 103
5.3.4 Embedded Deployment 105
5.3.5 Ablation Studies 106

6 DNN Mapping Optimization on Multi-Accelerator System on Chips 111

6.1 Related Works 113
6.2 Proposed Method 115
6.2.1 Mapping optimization strategy 116
6.2.2 SoC with Incompatible Data Formats 120
6.2.3 SoC with Specialized HW Units 122
6.3 Experimental Results 124
6.3.1 Setup 124
6.3.2 Search-Space Exploration 126
6.3.3 Additional Comparisons 129
6.3.4 Embedded Deployment 133
6.4 Implementation Details 136
6.4.1 DIANA 137
6.4.2 Darkside 139

7 Applications **141**

7.1 PPG-based Heart Rate Estimation on Wearables 143
7.1.1 Collaborative Heart Rate Inference System 144
7.1.2 Experimental Results 148
7.2 Visual-pose Estimation on Nano-Drones 152
7.2.1 Proposed Method 152
7.2.2 Experimental Results 154
7.2.3 In-field experimental evaluation 156
7.3 People-Counting on Low-Resolution Infrared Arrays 158
7.3.1 Proposed Method 159
7.3.2 Experimental Results 160

8 Conclusion **165**

References 167

List of Figures

1.1 Schematic structure of this manuscript. 5

2.1 Generation of the first two output time samples in a TCN layer with $K = 3$, $d = 4$, $F = 9$, and $s = 1$ for the first output channel ($m = 0$). . . 14

2.2 STM32H7 MCU block diagram [1]. 28

2.3 GAP8 SoC block diagram [2]. 29

2.4 MPIC block diagram [3]. 30

2.5 NE16 block diagram [4]. 31

2.6 DIANA block diagram [5]. 32

2.7 Darkside block diagram [6]. 33

2.8 HWatch block diagram [7]. 33

2.9 MAUPITI block diagram [8]. 34

3.1 Search space considered by PIT 40

3.2 Example of channels search. Each $\Theta_{A,m} = 0$ masks a slice of size $K \times C_{in}$ of the weights tensor W corresponding to the m-th convolutional filter. 41

3.3 Example of search of the receptive field. By zeroing out a time-slice of size $C_{out} \times C_{in}$ of the weights tensor W, each $\Theta_{B,i} = 0$ removes from the convolution output the contribution of 1 input time-step. . . 43

3.4 An example of a conversion between the binary masks Θ_B and Θ_Γ and the trainable architectural parameters β and γ for a layer with $F_{seed} = 9$. 44

3.5 Example of dilation search. d is increased by a factor 2 for every Γ_i = 0. 45

3.6 Comparison between seed, hand-tuned TCNs, and PIT Pareto fronts for the four target benchmarks. 53

3.7 Comparison of PIT search results with different search space definitions for PPG-DaLia. 56

3.8 Comparison of $\mathcal{R}_{size}$ and $\mathcal{R}_{ops}$ regularizers for PPG-DaLia. 56

3.9 Comparison in the Num. of Parameters vs MAE between PIT and SotA NAS and pruning tools on PPG-DaLia. 57

3.10 Comparison of search time and search space dimension between PIT and state-of-the-art NAS tools on PPG-DaLia. 59

4.1 Overview of the proposed joint quantization and pruning scheme for a DNN's generic layer. 66

4.2 Weights channels reordering by bit-width. 75

4.3 Comparison of different sampling methods for our proposed approach on CIFAR-10, GSC, and Tiny ImageNet. 77

4.4 Comparison of state-of-the-art approaches with the results obtained by our proposed method. 78

4.5 Comparison of model performance in the accuracy-versus-cycles space for various cost regularization strategies on CIFAR-10. The plot on the left shows results when targeting MPIC for deployment, while the plot on the right corresponds to models optimized for deployment on NE16. 80

4.6 Bit-width distributions of weight channels across three models trained on GSC using different strategies: PIT combined with MixPrec (P+M), our joint MixPrec and pruning approach (Ours), and MixPrec alone (M), all with size regularization applied. The input and output layers are labeled as L_0 and L_{out}, respectively, while DW and PW refer to depthwise and pointwise convolutional layers. 82

4.7 Weights bit-widths distributions on CIFAR-10 for models with different regularizers employed in the training objective and with different complexity. 83

4.8 Pareto fronts of architectures obtained on CIFAR-10 by either assigning the precision layer-wise from the set $P_X = \{2,4,8\}$ or quantizing the activations at 8 bits. 85

5.1 Target path-based DNAS method 90

5.2 Target mask-based DNAS method. 91

5.3 Results for different size targets in the Accuracy versus OPs. (Top row) Results obtained with mask-based DNAS algorithm. (Bottom row) Results obtained using a path-based DNAS algorithm. 101

5.4 Accuracy vs L2 memory violation. 104

5.5 Model size as objective (classical) vs. constraint (DUCCIO). 107

5.6 Comparison for different size targets of absolute value and max() constraints. 108

5.7 Comparison between the scheduling of λ proposed in Sec. 5.2.4 and a constant λ for DUCCIO's training. 109

5.8 Comparison between discretized Gumbel Softmax sampling and ICV Loss. 110

6.1 Alternative methods for executing a DNN on a heterogeneous SoC with shared memory. In Strategy A, all computations are carried out on a single CU. Strategy B assigns individual layers to different CUs. Strategy C, implemented in ODiMO, enables fine-grained intra-layer partitioning, where computations within a single layer are distributed across multiple CUs concurrently. 111

6.2 ODiMO mapping strategy of a layer on N different possible CUs. . 116

6.3 Re-organization pass to enable layer partitioning. 121

6.4 ODiMO mappings discovered when using as optimization target the latency. 126

6.5 ODiMO mappings discovered when using as optimization target the energy consumption. 128

6.6 (Top) Evaluation of ODiMO mappings obtained using structured channel pruning and exclusive execution on the digital compute unit of the DIANA platform for the CIFAR-10 task; (Bottom) Evaluation of ODiMO mappings obtained using a layer-wise differentiable neural architecture search strategy on the Darkside platform for the CIFAR-10 task. 129

6.7 Detailed per-layer comparison of latencies and of the ODiMO assignments for the “Ours" solution and Pruning ratios for the “Pr-l" and “Pr-m" solutions of Fig. 6.6-Top on the DIANA SoC; (A) Layer-level breakdown of CU assignments and channel pruning decisions; (B) Weighted average CU assignment and pruning across entire networks; (C) Per-layer cycle counts for each CU; (D) Total cycle count for each solution. 130

6.8 Equivalent of Fig. 6.7 for Fig. 6.6-Bottom on the Darkside SoC. . . 131

6.9 ODiMO mappings obtained using networks with different width multiplier and with latency as optimization target. 132

7.1 Components of the CHRIS framework. 144

7.2 CHRIS Decision Engine 146

7.3 Baseline models energy consumption (left) and average MAE on PPG-Dalia (right). 148
7.4 Configurations of CHRIS in the MAE vs. Energy space. 150
7.5 Pareto curves of the networks extracted using PIT in the clock cycles vs. MAE space (lower is better). 155
7.6 In-field control errors distribution (lower is better). Boxplot whiskers mark the 5th and 95th percentile of data. 157
7.7 Overview of the full-stack optimization flow. 158
7.8 Architecture and Precision Search Space exploration results. Different colors encode the different precisions' configurations. 161
7.9 Comparison of Pareto frontiers with and without post-processing. . 162
7.10 State-of-the-Art Comparison. 163

List of Tables

1.1 Summary of each chapter's different benchmarks and target architectures. 6

3.1 Range of regularizer strength (λ) values for the four benchmarks. . . 54

3.2 Deployment results on GAP8 and STM32 for the four considered benchmarks. 60

4.1 Average total training time speed-up of our approach with respect to the sequential application of PIT and MixPrec 81

4.2 Performance of baseline models quantized at fixed-precision and models trained with NE16 or MPIC regularizer on CIFAR-10. . . . 81

5.1 Deployment of ICL models on GAP8. 105

5.2 Deployment after layer-wise memory reduction of Tiny ImageNet models on GAP8. 105

6.1 Micro-benchmarking of modeled ResNet and MobileNet layers execution on DIANA and Darkside. 133

6.2 Deployment results of selected solutions from Fig. 6.4 on DIANA. . 135

7.1 Models used to build CHRIS configurations. 145

7.2 Configurations used within CHRIS. 145

7.3 Baseline models deployment on the Raspberry Pi3 and on the STM32WB55.148

7.4 Test set experiment results 156

7.5 In-field experiment results 156
7.6 Deployment results. 163

Chapter 1

Introduction

Deep Learning (DL) is becoming the go-to approach to solving previously unattainable complex problems in a data-driven fashion. For this reason, nowadays, Deep Neural Networks (DNNs) can be found at the backbone of a wide range of applications. A non-exhaustive list includes Computer Vision (CV) systems for autonomous navigation [9], object detection [10], and semantic segmentation [11]; Natural Language Processing (NLP) for virtual assistants [12], and translation services [13]; audio processing for speech recognition [14], and audio generation [15]; anomaly detection for structural health monitoring [16], and industrial maintenance [17]; bio-signals analysis to monitor health condition such as heart rate [18] or blood pressure [19].

Traditionally, the inference phase of these applications happens in powerful cloud resources. Nonetheless, there is a growing need to deploy DNNs directly on edge devices [20] such as Internet of Things (IoT) nodes and mobile devices. Edge deployment offers several advantages within four main directions, i.e., reduced latency, reduced energy consumption, improved privacy, and improved reliability [21]. Latency reductions are achieved by eliminating or reducing the need for network round-trips, where data are collected at the edge and transmitted to remote servers where the actual DNN inference happens. Many real-time applications, such as autonomous navigation [22, 9], cannot tolerate such delays, and therefore, executing inference at the edge is the only feasible alternative to enable them. Similarly, energy efficiency can be improved by avoiding transmitting data to the cloud and by performing computations locally [21]. Thus, local inference is preferable for

battery-operated devices, such as the vast majority of IoT nodes [23], in order to extend their lifetime and improve their sustainability. Enhanced privacy is ensured as data, which may be sensitive, remains on the user's device rather than being transmitted to external servers. This feature is of primary importance for applications such as biometric authentication [24] and personalized healthcare [19]. Finally, edge applications are inherently more reliable and can operate even when network connectivity is limited or unavailable.

Despite these benefits, deploying DL models on edge devices presents several challenges due to their inherent resource constraints [21]. Storage represents the first hard constraint of such platforms with values ranging from hundreds of kBs to a few MBs for Microcontroller Units (MCUs) such as STM32 [1]. This characteristic clashes with modern DNN designs, which often require hundreds of MB to store their parameters [25, 26, 10]. As an example, in Ch. 3 and Ch. 5 we will show how we can take hand-designed over-parametrized state-of-the-art DNNs and compress them by up to $8\times$ with no accuracy loss to respect the memory constraints of MCUs. Then, edge devices typically operate at lower clock frequencies and lack support for specialized instructions such as high-precision floating-point operations, commonly available in cloud-based hardware [27]. In Ch. 4 we will show how, thanks to quantization, we can reduce memory by up to $10\times$ with no task-performance loss compared to a floating-point baseline. Nonetheless, the current edge ecosystem is increasingly more heterogeneous and encompasses System-on-Chips (SoCs) with Computing Units (CUs) with diverse architectures [2] and specialized modules [6] that require specialized optimization approaches [28]. These CUs expose different trade-offs regarding latency, energy, and task accuracy. In Ch. 6 we will show how, by properly splitting the computational workload on the available CUs in heterogeneous SoCs, we can reduce the inference latency by up to $8\times$ with no accuracy drop compared to a solution using only the most accurate available CU. Finally, depending on the application [9, 29], many edge devices operate on batteries with strict power budgets, necessitating highly optimized energy-efficient inference.

This thesis addresses the challenge of enabling efficient DL inference on resource-constrained edge devices. In particular, the techniques presented in this work are built on three pillars: **design automation**, **hardware-awareness**, and **training requirements democratization**.

Design Automation

Current DNN architectures are typically designed manually following rules of thumb [25, 30]. Similarly, practical adoption of optimization techniques is limited by the required manual effort and expertise [27, 31]. Given the tighter constraints and the required domain knowledge, this problem is exacerbated when targeting the deployment of DNNs on severely resource-constrained hardware (HW). Automating the optimization process is crucial to ease the need for direct human intervention, speeding up and allowing scale DNN deployment over a wide range of hardware platforms and tasks. For these reasons, all the optimization techniques proposed in this thesis aim to develop an automated optimization pipeline. This vision is enforced in the open-source PLiNIO [1] (Plug-and-play Lightweight Neural Inference Optimization) library, which has been developed during this PhD. In particular, PLiNIO represents an attempt to engineer the main techniques discussed in this manuscript in a comprehensive framework that abstracts away the main complexities. PLiNIO aims to automatically identify model components for optimization, adapt to different DNN architectures without human intervention, and generate deployment-ready optimized models with minimal user input. Please note that the main discussion of this thesis will be about the developed techniques and their scientific outcomes and will be less focused on the engineering part required to build the framework. Further details about the internals of PLiNIO are available in [32].

Hardware-awareness

Generic optimization techniques often yield suboptimal results when deployed on specific hardware [33–35]. This thesis presents novel hardware-aware optimization methods that leverage hardware-agnostic metrics (e.g., number of parameters, number of operations, etc.) and a detailed knowledge of target hardware capabilities and limitations. In this way, the hardware details (e.g., type of computing units, memory hierarchy, etc.) directly enter and guide the optimization process, offering tailored solutions on the target devices.

[1] https://github.com/eml-eda/plinio

Training Requirements Democratization

Current optimization strategies typically require substantial computational resources for the optimization process itself [36–38]. The work discussed in this thesis is focused on democratizing access to optimization by reducing the computational requirements for performing model optimization. In particular, the main target of the developed techniques is keeping the optimization cost, in terms of required time and memory, comparable with the one of a single training run of the DNN. The rationale behind this choice is enabling optimization on consumer-grade GPUs rather than specialized server-grade accelerators. On the one hand, this decreases the time and the cost required to optimize models. On the other, it makes optimization accessible to researchers and developers with limited resources.

1.1 On the Absence of Attention-based Transformers

The research project described in this thesis started at the end of 2021. At that time, transformer-based architectures [39, 26] were rarely considered for ultra-low-power deployments, whereas convolutional and fully-connected (FC) networks were already supported by deployment tool-chains on MCUs. Accordingly, the present work restricts its contributions to convolutional and FC operators.

Additionally, in a standard transformer block, the feed-forward module, composed of FC layers and typically placed after attention, accounts for roughly two-thirds of the total parameters [40]. Consequently, the compression, pruning, and mixed-precision techniques developed and discussed in the main body of this thesis can be used directly on those sub-components.

Finally, it is important to point out that ideas similar to the ones presented in this thesis have been applied to optimize the attention module of transformers. E.g., [41] proposes a pruning approach for Vision Transformers, with similarities to the one that will be presented in this thesis in Ch. 3, or [42] explores the usage of mixed-precision search for language models as we will present for Convolutional architectures in Ch. 4.

1.2 Thesis Structure

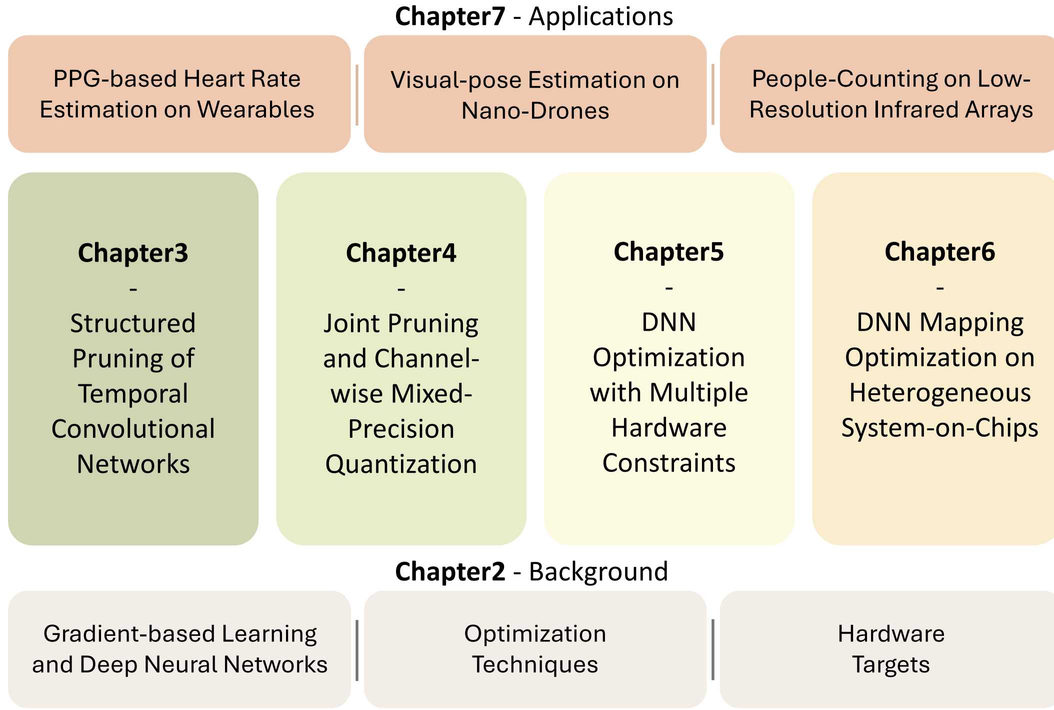

Fig. 1.1 Schematic structure of this manuscript.

The structure of this thesis is summarized in Fig. 1.1, while Table 1.1 summarizes the different benchmarks and architectures used in the main body of the manuscript (i.e., Ch 3, Ch 4, Ch 5, and Ch 6). Please note that the choices of the considered architectures follow the choice of the respective benchmarks, which have been selected to be aligned with the state-of-the-art and the benchmarked techniques and platforms. Within the limits of each technique (discussed in each chapter), different choices of benchmarks and architectures are possible.

Ch. 2 provides the necessary background knowledge by introducing the main types of DNNs considered, reviewing gradient-based learning and general DNN optimization techniques such as Pruning, Quantization, Neural Architecture Search (NAS), and knowledge distillation. The chapter also includes a discussion of the major Deep Learning frameworks and their structure. An overview of the hardware platforms targeted in this work concludes the chapter.

Ch. 3 presents PIT, a structured pruning algorithm designed to optimize Temporal Convolutional Networks for time-series analysis on edge devices.

Table 1.1 Summary of each chapter's different benchmarks and target architectures.

	Benchmarks	Architectures
Ch. 3	PPG-Dalia [43]	TEMPONet [46]
	ECG5000	ECGTCN [47]
	NinaPro DB1 [44]	TCCNet [48]
	GSC v2 [45]	TC-ResNet14 [49]
Ch. 4	CIFAR-10 [50]	ResNet-8 [52]
	GSC v2 [45]	DS-CNN [14]
	Tiny ImageNet [51]	ResNet-18 [25]
Ch. 5	CIFAR-10 [50]	ResNet-8 [52]
	GSC v2 [45]	DS-CNN [14]
	VWW [53]	MobileNetV1 [30]
	DCASE2020 [54]	FC-Autoencoder [54]
	Tiny ImageNet [51]	ResNet-18 [25]
Ch. 6	CIFAR-10 [50]	ResNet-20 [25] and MobileNetV1 [30]
	CIFAR-100 [50]	ResNet-18 [25] and MobileNetV1 [30]
	ImageNet [55]	ResNet-18 [25] and MobileNetV1 [30]

Ch. 4 explores the integration of mixed-precision quantization with pruning in a comprehensive framework to optimize Convolutional Neural Networks.

Ch. 5 extends the discussion by adapting pruning and NAS to optimize DNNs under multiple hardware constraints simultaneously.

Ch. 6 presents a mapping optimization technique that enables efficient inference on heterogeneous SoCs by distributing computations across multiple computing units.

Ch. 7 presents some applications of the proposed methods on real-world tasks. These tasks include PPG-based heart rate estimation on wearable devices, visual pose estimation on nano-drones, and privacy-preserving people counting using low-resolution infrared arrays.

Finally, Ch. 8 concludes the thesis by summarizing the main achievements of the discussed works.

Please note that throughout the whole manuscript, I will consistently use the pronoun “we" rather than “I" when referring to the work presented. Although this thesis is an individual submission, the research it contains is the result of collaboration with my supervisors and colleagues. The use of “we” acknowledges their contributions and reflects the collective nature of the work conducted.

1.3 List of Publications

The following list details the papers published in peer-reviewed international journals and conferences on which this thesis is based:

1. Burrello, Alessio, Daniele Jahier Pagliari, **Matteo Risso**, Simone Benatti, Enrico Macii, Luca Benini, and Massimo Poncino. "Q-ppg: Energy-efficient ppg-based heart rate monitoring on wearable devices." IEEE Transactions on Biomedical Circuits and Systems 15, no. 6 (2021): 1196-1209.

2. **Risso, Matteo**, Alessio Burrello, Daniele Jahier Pagliari, Francesco Conti, Lorenzo Lamberti, Enrico Macii, Luca Benini, and Massimo Poncino. "Pruning in time (PIT): A lightweight network architecture optimizer for temporal convolutional networks." In 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 1015-1020. IEEE, 2021.

3. Burrello, Alessio, Daniele Jahier Pagliari, Pierangelo Maria Rapa, Matilde Semilia, **Matteo Risso**, Tommaso Polonelli, Massimo Poncino, Luca Benini, and Simone Benatti. "Embedding temporal convolutional networks for energy-efficient ppg-based heart rate monitoring." ACM Transactions on Computing for Healthcare (HEALTH) 3, no. 2 (2022): 1-25.

4. **Risso, Matteo**, Alessio Burrello, Francesco Conti, Lorenzo Lamberti, Yukai Chen, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. "Lightweight neural architecture search for temporal convolutional networks at the edge." IEEE Transactions on Computers 72, no. 3 (2022): 744-758.

5. **Risso, Matteo**, Alessio Burrello, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. "Multi-complexity-loss DNAS for energy-efficient and memory-constrained deep neural networks." In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, pp. 1-6. 2022.

6. **Risso, Matteo**, Alessio Burrello, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. "Channel-wise mixed-precision assignment for dnn inference on constrained edge nodes." In 2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC), pp. 1-6. IEEE, 2022.

7. Burrello, Alessio, **Matteo Risso**, Noemi Tomasello, Yukai Chen, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. "Energy-efficient wearable-to-mobile offload of ML inference for PPG-based heart-rate estimation." In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1-6. IEEE, 2023.

8. Cereda, Elia, Luca Crupi, **Matteo Risso**, Alessio Burrello, Luca Benini, Alessandro Giusti, D. Jahier Pagliari, and Daniele Palossi. "Deep neural network architecture search for accurate visual pose estimation aboard nano-uavs." In 2023 IEEE International Conference on Robotics and Automation (ICRA), pp. 6065-6071. IEEE, 2023.

9. **Risso, Matteo**, Alessio Burrello, Giuseppe Maria Sarda, Luca Benini, Enrico Macii, Massimo Poncino, Marian Verhelst, and Daniele Jahier Pagliari. "Precision-aware latency and energy balancing on multi-accelerator platforms for dnn inference." In 2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), pp. 1-6. IEEE, 2023.

10. Pagliari, Daniele Jahier, **Matteo Risso**, Beatrice Alessandra Motetti, and Alessio Burrello. "Plinio: a user-friendly library of gradient-based methods for complexity-aware DNN optimization." In 2023 Forum on Specification & Design Languages (FDL), pp. 1-8. IEEE, 2023.

11. Burrello, Alessio, **Matteo Risso**, Beatrice Alessandra Motetti, Enrico Macii, Luca Benini, and Daniele Jahier Pagliari. "Enhancing neural architecture search with multiple hardware constraints for deep learning model deployment on tiny IoT devices." IEEE Transactions on Emerging Topics in Computing 12, no. 3 (2023): 780-794.

12. **Risso, Matteo**, Francesco Daghero, Beatrice Alessandra Motetti, Daniele Jahier Pagliari, Enrico Macii, Massimo Poncino, and Alessio Burrello. "Optimized Deployment of Deep Neural Networks for Visual Pose Estimation on Nano-drones." In European Robotics Forum, pp. 304-309. Cham: Springer Nature Switzerland, 2024.

13. **Risso, Matteo**, Chen Xie, Francesco Daghero, Alessio Burrello, Seyedmorteza Mollaei, Marco Castellano, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. "HW-SW Optimization of DNNs for Privacy-Preserving People

Counting on Low-Resolution Infrared Arrays." In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1-6. IEEE, 2024.

14. Motetti, Beatrice Alessandra, **Matteo Risso**, Alessio Burrello, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. "Joint Pruning and Channel-wise Mixed-Precision Quantization for Efficient Deep Neural Networks." IEEE Transactions on Computers (2024).

15. **Risso, Matteo**, Alessio Burrello, and Daniele Jahier Pagliari. "Optimizing DNN Inference on Multi-Accelerator SoCs at Training-time." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2025).

Chapter 2

Background

This chapter provides the necessary background knowledge. First, it describes in Sec. 2.1 the main types of DNNs considered, i.e., 2D Convolutional Neural Networks (CNNs) and Temporal Convolutional Networks (TCNs). Then, it provides an overview of gradient-based learning as the primary strategy for the parameters' optimization method of DNNs. After that, regularization techniques are introduced, which serve as the foundation for encoding cost metrics optimization in later chapters. The section concludes with an overview of the main Deep Learning frameworks available highlighting their major components as autograd and how these have been crucial in making Deep Learning a pervasive technique. Sec. 2.2 covers the basic concepts of general DNN optimization techniques such as pruning, quantization, NAS, and knowledge distillation. Sec. 2.3 concludes the chapter with an overview of the hardware platforms considered in this work.

2.1 DNNs and Gradient-based Learning

2.1.1 Deep Neural Networks

In many problems, we have access to input data $\mathbf{x} \in \mathbb{R}^n$ from which we want to compute a function $f : \mathbb{R}^n \to \mathbb{R}^p$ that extracts meaningful quantities from these inputs. However, in most real-world scenarios, the explicit form of f is unknown, and we instead approximate it using a parametric model $\hat{f}_\theta$, where θ denotes a set of parameters to be learned from data. DNNs are a class of such function approximators that leverage stacked compositions of simple transformations to model complex relationships in the input data.

Multi Layer Perceptrons

The basic building block of such DNNs is represented by the so-called perceptron [56]. A perceptron is a parametric function that maps an input vector $\mathbf{x} \in \mathbb{R}^n$ to a scalar output $y \in \mathbb{R}^n$ through a weighted linear combination followed by a non-linear activation. This is expressed as:

$$y = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right) \tag{2.1}$$

Where $\mathbf{w}$ is known as the weight vector, $b \in \mathbb{R}$ is the bias term and $\sigma : \mathbb{R} \to \mathbb{R}$ is the so-called activation function which is necessary to express non-linear mappings between the input and the output spaces.

The second key idea for the construction of DNNs is to extend the representational capacity of the perceptron in the width dimension by considering a layer composed of several perceptrons acting in parallel. Each unit in the layer computes its function on the same input vector. If a layer consists of m output features, the computation can be vectorized as:

$$\mathbf{z} = \sigma\left(W\mathbf{x} + \mathbf{b}\right) \tag{2.2}$$

Where $W \in \mathbb{R}^{m \times n}$ is the weight matrix with each row corresponding to individual perceptrons weight vector, $\mathbf{b} \in \mathbb{R}^m$ is the bias vector, and the activation σ is applied element-wise. This formulation enables the layer to output a vector $\mathbf{z} \in \mathbb{R}^m$, mapping the input into a new feature space. These layers of multiple units are commonly denoted as *fully-connected* (FC), *dense*, or *linear* layers. By stacking multiple layers, we obtain the so-called Multi-Layer Perceptron (MLP), a composition of functions

that transforms the input representation at each stage. Formally, an MLP with L layers defines a function:

$$f(\mathbf{x}) = f^{(L)}\left(f^{(L-1)}\left(\cdots f^{(1)}(\mathbf{x})\cdots\right)\right) \tag{2.3}$$

where each intermediate layer, commonly denoted as *hidden*, follows the structure of Eq. 2.2, i.e.,:

$$\mathbf{z}^{(l)} = \sigma^{(l)}\left(W^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad \text{with} \quad \mathbf{z}^{(0)} = \mathbf{x} \tag{2.4}$$

Here, $W^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector for layer l, and $\sigma^{(l)}$ is again the activation function. The composition of these transformations allows MLPs to approximate highly non-linear functions by learning hierarchical representations of the input data [56].

The equation implemented by the final layer of an MLP depends on the problem being solved. In regression tasks, the output is typically a linear function of the last hidden layer:

$$\hat{y} = W^{(L)}\mathbf{z}^{(L-1)} + \mathbf{b}^{(L)} \tag{2.5}$$

For classification tasks, the output is a probability distribution over C possible classes obtained using a non-linear activation such as the *softmax*:

$$\hat{y}_j = \frac{\exp\left(z_j^{(L)}\right)}{\sum_k \exp\left(z_k^{(L)}\right)}, \quad j = 1, \ldots, C \tag{2.6}$$

The pre-softmax outputs of a DNN are commonly reffered as *logits*.

We define an MLP as a DNN when it has more than one hidden layer, i.e., $L > 2$.

Various activation functions have been proposed, with the *sigmoid* function being one of the earliest and most widely used. The sigmoid function smoothly maps inputs to the bounded range $[0,1]$, making it useful for probabilistic interpretation. However, it suffers from issues such as *vanishing gradients* (see Sec. 2.1.2), which can hinder training in deep networks [57]. In contrast, modern deep learning architectures predominantly use the *Rectified Linear Unit (ReLU)*, defined as: $\text{ReLU}(x) = \max(0, x)$. ReLU is computationally simple yet highly effective, mitigating the vanishing gradient problem and enabling faster and more stable convergence during training [57].

Due to its advantages, ReLU has become the standard activation function in deep neural networks.

The Universal Approximation Theorem [58] states that any continuous function on a subset of $\mathbb{R}^n$ can be approximated by a sufficiently wide MLP with a single hidden layer. However, deeper networks, thus leveraging hierarchical representations, can often achieve better results with fewer parameters [56]. For all these reasons, DNNs are a powerful class of function approximators capable of learning complex mappings from data in various scenarios.

While MLPs represent the first architecture used as a function approximator, their application as a standalone model has become less common. Instead, they are primarily found as components of more complex DNN architectures. In particular, MLPs are often used as the final stage of DL models. Conversely, the earlier layers focus on extracting structured features from raw input data. Finally, the MLP component integrates these features to compute the final output. This design pattern can be found in architectures such as CNNs, which represent the focus of the next Sec. 2.1.1.

Convolutional Neural Networks

CNNs are DNN architectures tailored for processing multi-dimensional data, such as images. CNNs represent today the de-facto standard in CV tasks such as object detection [10] and image classification [50]. They implement feature extraction by applying the *convolution* operation, which computes local weighted sums over the input using learnable filters. The inputs and outputs of convolution are usually denoted as the *input* and *output feature map* or *activations*. For a single-channel, two-dimensional input feature map x of size $t \times t$, a convolutional layer applies a $k \times k$ filter K to produce an output feature map y according to:

$$y_{(m,n)} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K_{(i,j)} x_{(m \cdot s+i-p, n \cdot s+j-p)}, \tag{2.7}$$

where s is the stride and p is the padding. The spatial dimension d of the output feature map is given by $d = \frac{t+2p-k}{s} + 1$. Similarly to MLPs, an activation function σ is computed on all the elements of y.

For multi-channel inputs, the convolution generalizes naturally. Let $x^{(c_{in})}$ denote the c_{in}-th input channel and $y^{(c_{out})}$ the c_{out}-th output channel. Then,

$$y^{(c_{out})}_{(m,n)} = \sum_{c_{in}=0}^{C_{in}-1} \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K^{(c_{in},c_{out})}_{(i,j)} \, x^{(c_{in})}_{(m \cdot s+i-p, n \cdot s+j-p)}. \tag{2.8}$$

As mentioned before, the early layers of a CNN are devoted to feature extraction from raw data. In contrast, the final layers are usually implemented as fully connected layers, which combine these features to compute the final output for tasks such as classification or regression [25]. Standard feature extractors include a variable number of convolutional layers in the form of Eq. 2.8 interleaved with Batch Normalization (BN) layers, which standardize the activations' statistic, stabilize the training, and improve convergence by reducing internal covariance shift [59]. Moreover, downsampling layers, typically denoted as pooling layers, are included to progressively reduce the feature map spatial dimension and improve translation invariance [50].

Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs) are one-dimensional convolutional

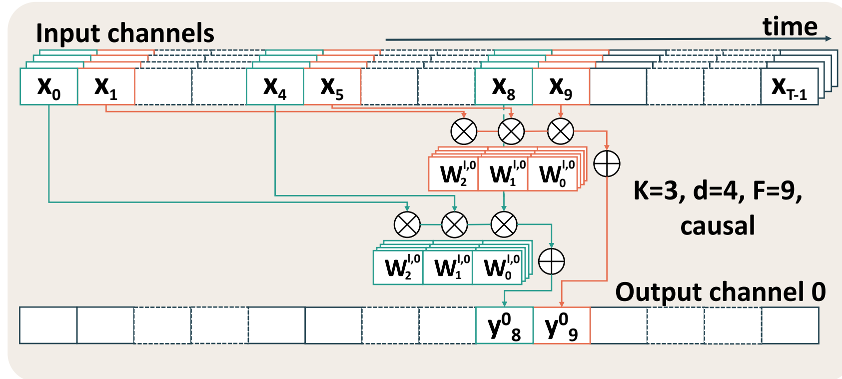

Fig. 2.1 Generation of the first two output time samples in a TCN layer with K = 3, d = 4, F = 9, and s = 1 for the first output channel (m = 0).

neural networks designed for time-series processing. They have recently gained significant attention due to their ability to achieve state-of-the-art results in various tasks [60, 49, 48]. Previous to the introduction of TCNs, such tasks were mainly

addressed by recurrent neural networks (RNNs) and their variants—such as Long-Short Term Memory (LSTM) and Gated Recurrent Units (GRU). Such architectures are not considered in this thesis, given that TCNs represent a strong alternative as they are less affected by issues like vanishing or exploding gradients and the high memory requirements of RNNs on long sequences. Furthermore, TCNs inherit the data locality and arithmetic intensity of standard CNNs, leading to latency- and energy-efficient inference [61].

The building blocks of TCNs are the same as those of standard CNNs, including convolutional, pooling, and fully-connected layers. However, the convolutional layers in a TCN are specifically adapted to temporal inputs by enforcing two key properties: *causality* and *dilation*. Causality ensures that the output y_t at time t depends only on the current and past inputs, thus preserving the natural order of events. Dilation introduces a fixed step d between the input samples processed by each filter, effectively enlarging the receptive field without increasing the number of parameters or computational cost. For a 1D input with C_{in} channels, the dilated convolution operation is defined as:

$$y_t^m = \sum_{i=0}^{K-1} \sum_{l=0}^{C_{in}-1} x_{ts-d\,i}^{l} \cdot W_i^{l,m}, \quad \forall m \in \{0,\dots,C_{out}-1\},\, \forall t \in \{0,\dots,T-1\} \tag{2.9}$$

Where K is the filter size, s the stride, $W_i^{l,m}$ the filter weights, and the receptive field is given by $F = d \cdot (K-1) + 1$. Fig. 2.1 depicts the structure of a TCN convolutional layer.

The first type of TCN proposed in literature [61] was a fully convolutional network with residual connections. Nonetheless, in the recent SotA [46], TCNs can also incorporate additional layers, such as strided convolutions, pooling, and fully connected layers, to further improve their performance

Recap of Key DNN Hyperparameters

DNN architectures' design generally involves many hyperparameters that determine model capacity, efficiency, and overall performance. For MLPs, key hyperparameters include the number of hidden layers (i.e., depth) and the number of output features in each layer (i.e., width). In CNNs, additional hyperparameters arise from the convolutional layers, such as the number of filters, kernel sizes, stride values, padding

schemes, pooling operations, and normalization parameters. TCNs further extend this set by introducing hyperparameters related to temporal processing, including the dilation factor d, which, together with the kernel size K, defines the receptive field $F = d \cdot (K-1)+1$.

The whole design space of DNNs, which is controlled by the above-mentioned hyper-parameters, is enormous and impossible to explore exhaustively manually. The technique presented in this thesis offers an automated, cost-aware approach for navigating this huge design space while tailoring DNNs to the underlying target hardware.

2.1.2 Gradient-based Learning

In previous Sec. 2.1.1, the functional forms of DNNs, including MLPs, CNNs, and TCNs, have been introduced. A set of parameters, denoted as θ, parameterizes these functions. This section details the process of selecting these parameters starting from data. This process is referred to as *learning*. While many learning paradigms exist, such as self-supervised and unsupervised learning, this work addresses the so-called supervised learning scenario.

The paradigm that aims to learn a mapping from input data to corresponding outputs is known as supervised learning. This is achieved by training a model on a dataset of labeled examples, where each example consists of an input and its associated target output (i.e., the label). During this procedure, the model learns to predict the output for new, never-seen-before inputs by generalizing from the patterns observed in the training data.

The training of DNNs is a critical step, as model performance is highly dependent on the quality of the learned parameters. Due to the inherent complexity and non-linearity introduced by activation functions, optimizing θ is a non-convex problem [56]. Consequently, a closed-form solution for this optimization does not exist. Then, the training process is formulated as a non-convex optimization problem, where the objective is to minimize a loss function, $\mathcal{L}(f(X,\theta),Y)$. This loss function quantifies the discrepancy between the network's predicted output $\hat{Y} = f(x;\theta)$ and the desired output Y, and its specific form is determined by the task at hand.

Given the non-convex nature of $\mathcal{L}(\theta)$, DNNs are trained using iterative gradient-based techniques that aim to drive the loss function to a low value. Stochastic

Gradient Descent (SGD) [62] and its variants form the basis of these techniques. The fundamental idea of gradient-based optimization is to iteratively update θ in the direction that minimizes $\mathcal{L}(\theta)$, which is dictated by the gradient $\nabla_\theta \mathcal{L}(\theta)$, leading to the following update rule:

$$\theta^{new} = \theta^{old} - \eta \nabla_\theta \mathcal{L}(\theta) \tag{2.10}$$

Where η is the learning rate, a critical hyper-parameter that controls the step size of the updates, an excessively small learning rate can lead to slow convergence. In contrast, an excessively large learning rate can cause divergence.

Plain gradient descent, or batch gradient descent, computes the gradient using the entire training dataset (X, y) at each iteration. This approach becomes computationally prohibitive for large datasets. SGD addresses this issue by approximating the gradient using a single randomly selected training instance or, more commonly, a mini-batch of instances. This stochastic approach introduces variance into the optimization process, causing the loss to fluctuate. However, this variance can also help the algorithm escape local minima, potentially leading to better generalization [56]. The size of the mini-batch is another hyper-parameter that requires tuning.

Beyond basic SGD, modern optimization algorithms have been developed to enhance convergence and performance. These algorithms, such as SGD with momentum [63] and Adam [64], start from plain SGD and incorporate techniques to adapt the learning rate and utilize past gradient information. For example, momentum accelerates learning in the relevant direction [63], while Adam adapts the learning rate for each parameter based on estimates of the first and second moments of the gradients [64]. In this thesis, we will specify, whenever needed, the specific employed optimization algorithm.

Regularization Techniques

In the previous section, we introduced the fundamental concepts of gradient-based learning for DNNs. However, a crucial general aspect of ML and DL, i.e., *generalization*, was not explicitly addressed. Generalization refers to the ability of a trained model to perform well on unseen data beyond the training set. Poor generalization can manifest as either overfitting or underfitting. **Overfitting** occurs when a model memorizes the training data. An overfitted model performs excellently on the train-

ing set but poorly on unseen data. **Underfitting**, on the other hand, occurs when a model fails to capture the underlying patterns in the training data, resulting in poor performance on both the training and unseen data.

Underfitting can be mitigated by ensuring the selected model is sufficiently expressive, i.e., with enough parameters θ. Conversely, overfitting can be reduced by employing *regularization* [65]. Regularization techniques introduce constraints or penalties into the learning process, helping to reduce the complexity of the learned model and preventing it from memorizing noise in the training data. Among the most common regularization techniques are those based on norm penalties [65], which are added directly to the task-specific loss function $\mathcal{L}$. Two of the most common examples are:

- **L^2-Regularization (Ridge Regularization):** This technique adds the squared L^2 norm of the parameter vector θ to the loss function:

  $$L^2(\theta) = \lambda \|\theta\|_2^2 = \lambda \sum_i \theta_i^2$$

  Geometrically, L^2-regularization corresponds to adding a circular or spherical constraint to the parameter space. It encourages the model to have small weights, effectively shrinking the parameter values towards zero [65]. In terms of gradient updates, L^2 regularization adds a term proportional to θ to the gradient, effectively pushing the parameters towards zero.

- **L^1-Regularization (LASSO Regularization):** This technique adds the L^1 norm of the parameter vector θ to the loss function:

  $$L^1(\theta) = \lambda \|\theta\|_1 = \lambda \sum_i |\theta_i|$$

  Geometrically, L^1-regularization adds a diamond-shaped constraint to the parameter space. It tends to drive some parameter values to zero, resulting in sparse weight vectors [66]. This property makes LASSO regularization useful for feature selection and model compression, as it effectively identifies and removes less important features. In terms of gradient updates, L^1 regularization adds a constant term to the gradient, pushing the parameters towards zero or away from zero, depending on their sign. Regularizers based

on LASSO, also known as sparsifying regularizers, are at the base of many pruning techniques [66].

For both types of regularization (L^2 and L^1), the parameter λ controls the strength of the regularization. A more regularized model is obtained when a stronger penalty is imposed on large parameter values associated with larger values of λ.

This thesis will use regularization techniques as the primary strategy to bias the optimization towards low-complexity solutions.

2.1.3 Deep Learning Frameworks

The rapid advances in DNN research over the last years was enabled by the confluence of new and powerful hardware such as GPUs coupled with the appearance of software stacks capable to abstract away hardware details and technicalities while retaining computational efficiency. Among these, PyTorch[67], TensorFlow[68], and JAX [69] represents some of the most used libraries by both academia and industry. Each of these libraries revolve around a similar core abstraction represented by multi-dimensional tensor and presents a concise Python frontend that hides the boundaries between CPUs, GPUs and specialised accelerators while still allowing to perform efficient computations on them.

Conceptually the runtime of every major framework can be decomposed into three tightly coupled layers. At the top, a Python front-end offers an imperative API that mirrors `NumPy`, so models can be expressed with familiar index-based notation and ordinary control-flow statements. The code written at this level is intercepted and lowered to an intermediate representation (IR): PyTorch captures graphs with FX or TorchScript, JAX produces XLA High Level Optimiser (HLO) programs, and TensorFlow relies on concrete Tensorflow Graph objects obtained by tracing eager code. This IR forms the corpus on which language-level transformations operate. Such transformations includes operation such as automatic differentiation, vectorization, partial evaluation, or data sharding. In the final stage the transformed program is either compiled ahead of time or dispatched at run time to highly optimized kernels written in C++, CUDA, or ROCm. When beneficial, vendor libraries such as cuDNN, or cuBLAS are invoked, otherwise kernels can be fused in chains of element-wise operations in order to reduce memory traffic and kernel-launch latency.

Historically, frameworks adopted mutually exclusive execution paradigms: *eager* systems such as early PyTorch constructed the computation graph on the fly, a choice that provided maximal flexibility for data-dependent shapes at the cost of modest Python overhead, whereas the original release of TensorFlow forced users to assemble a static graph up-front and then execute it multiple times to exploit compiler-level optimisation. Today the distinction has blurred. PyTorch2 introduces an ahead-of-time compiler that traces a dynamic program, rewrites it as a static IR and emits a graph executable, while TensorFlow2 defaults to eager execution and only switches to graph mode when a function is decorated with `tf.function`. JAX follows a similar path by staging pure Python functions through the `@jit` decorator so that they materialise as HLO before reaching the target device.

All the work described in this thesis is based on Pytorch 2.x version.

Autograd

One of the key element of such DL frameworks is represented by the Autograd (AD) module which implements efficient computation of gradients which is crucial for scaling gradient-based learning. The backpropagation algorithm [62] and its implementations within automatic differentiation software libraries [67] provide an efficient method for computing $\nabla_\theta \mathcal{L}(\theta)$ by performing two passes through the network: a forward pass and a backward pass. During the forward pass, the mini-batch is propagated through the network, and the loss is computed. During the backward pass, the gradients are calculated using the calculus chain rule, starting from the output layer and propagating backward to the input layer. These gradients are then used to update θ according to the update rule of Eq. 2.10.

In particular, modern frameworks implement AD as a source-to-source transform applied to the IR introduced earlier. PyTorch employs AOT-Autograd, JAX rewrites `jaxpr` graphs through `grad`, and TensorFlow generates a companion graph that contains the vector-Jacobian product (VJP) of each primitive op. Since the transform is purely functional, it composes seamlessly with other passes such as vectorization.

2.2 Optimization Techniques

2.2.1 Pruning

Pruning is a fundamental optimization technique for compressing DNNs and improving inference speed [66] which involves reducing the model size by selectively removing parameters. Pruning methods can be categorized based the selection strategy used to determine which parameters to eliminate and on the granularity of the removed elements.

From a granularity perspective, pruning approaches can be classified as *unstructured* or *structured*. Unstructured pruning removes individual weights, offering the highest theoretical reduction in parameter count and computational cost without significant accuracy loss, with reported reductions of up to 90% [70]. However, the resulting sparse weight matrices lead to irregular memory access patterns and reduced hardware utilization, particularly on general-purpose platforms, where additional overhead is required to properly exploit sparsity [71, 66]. Sparse computations also affect cache performance and make parallel execution less efficient [71]. Conversely, unstructured pruning can be properly exploited when specific hardware and software support is provided [40].

Structured pruning removes weights more regularly, such as in blocks or entire neurons, channels, or filters. While block-based pruning still requires specialized hardware support to accelerate execution efficiently [66], channel- and filter-wise pruning result in compressed network architectures that can be transformed into smaller, dense models using distributive and associative algebraic properties [66]. This eliminates the deployment challenges of unstructured pruning, as no special hardware or software support is required. However, structured pruning typically results in a worse trade-off between compression and task performance [70].

Various selection schemes have been proposed to determine which parameters to prune. Magnitude-based pruning removes weights with the smallest absolute values and is a simple yet effective method [70, 72]. It applies to individual weights and structured groups [72] and does not require additional data. However, aggressive pruning may degrade performance, necessitating fine-tuning and potentially diminishing the benefits of a data-free approach [66]. Sensitivity-based pruning addresses this issue by leveraging training data to assess the impact of removing specific parameters on the model's output [73]. However, these methods typically operate on a pre-trained network (potentially followed by fine-tuning) and do not

exploit the benefits of jointly training and pruning weights [66]. Gradient-based pruning overcomes this limitation by integrating pruning into the training process. This is often achieved by introducing trainable binary gates that determine which weights or structures should be retained [31, 34]. These gates and the network's weights are jointly optimized using gradient descent to minimize a loss function that combines the primary task objective with a regularization term related to model size or computational cost.

The pruning techniques presented in Ch. 3 of this thesis belong to the family of gradient-based structured pruning where learnable binary masks are used to selectively remove parts of the weight tensors W corresponding to different hyper-parameters configuration of the DNN under optimization. This particular pruning strategy is akin to the family of mask-based Differentiable Neural Architecture Search algorithms presented in the next Sec. 2.2.2.

2.2.2 Neural Architecture Search

In recent years, Neural Architecture Search (NAS) has gained significant traction in academia and industry as a powerful technique for automating the design of DL models. Several surveys [74–76] have attempted to summarize the rapidly evolving landscape of NAS methodologies, categorizing them along three main dimensions: the search space, the search strategy, and the evaluation strategy.

The search space defines the set of potential architectural configurations that a NAS method can explore. This has been the subject of many different proposals in the literature. Typical search spaces range from coarse-grained global hyperparameter tuning, such as determining the number of layers or global width multipliers, as seen in efficientnet [77] and mobilenetv3 [78], to cell-based approaches where the number and type of cells are not predetermined [36, 79]. In this case, each cell represents a specific combination of one or more layers. Other methods employ fine-grained layer-wise tuning, adjusting hyperparameters like the number of channels, filter size, and dilation rates [34, 31]. The number of possible architectures within these search spaces can be huge, reaching up to 10^{32} alternatives [34]. Within the search space, each configuration is uniquely represented by an encoding scheme specific to the NAS method and closely related to the search strategy.

The search strategy defines the optimization mechanism used to explore the search space and to select high-quality architectures. While achieving high task performance, such as classification accuracy or regression error, is always the primary objective, modern hardware-aware NAS methods incorporate non-functional constraints, including memory footprint, computational complexity, and latency [76].

Finally, the evaluation strategy determines how candidate architectures are scored during the search. The most direct approach for predictive performance is full training and testing on a reference dataset. Similarly, non-functional metrics can be obtained by deploying the model on target hardware [37]. However, both methods are computationally expensive, justifying the development of proxy-based evaluation techniques.

NAS search strategies can be broadly classified into two main families: black-box and one-shot approaches. Early NAS methods primarily relied on black-box optimization techniques, such as Reinforcement Learning (RL) [36], and Evolutionary Algorithms (EA) [38]. These methods iteratively sample one or more architectures from the search space, evaluate them according to functional and non-functional metrics, update the sampling policy based on the results, and repeat the process. Black-box strategies are highly flexible and can accommodate various constraints and objectives. However, their reliance on repeated sampling, training, and evaluation makes them computationally prohibitive, often requiring thousands of GPU hours for a single search [37].

To reduce the computational cost of black-box NAS, proxy-based evaluation strategies have been introduced. Predictive performance can be estimated from early-stage learning curves, testing on a small subset of minibatches, or training for only a few epochs [74, 80]. Similarly, non-functional metrics can be approximated using analytical models or hardware simulators [76]. However, accuracy extrapolation is unreliable, and even with proxy-based techniques, black-box NAS remains computationally demanding, limiting its accessibility for resource-constrained IoT system designers [35].

To address these limitations, a class of more efficient techniques, known as Differentiable NAS (DNAS) or One-shot NAS, has been introduced [79]. These methods, which are the primary focus of this thesis, aim to significantly reduce the computational cost of NAS while maintaining competitive performance. They are discussed in detail in the following section.

Differentiable Neural Architecture Search

DNAS methods employ gradient descent as a search strategy, where within the same training loop, the network weights and architecture are jointly optimized. This characteristic, often referred to as *one-shot* training, strongly couples the search and evaluation strategies.

The first DNAS techniques were based on the *supernet* concept, which are DNNs incorporating different paths, each corresponding to a candidate design point in the search space [79]. A supernet is constructed by replacing each layer of a reference DNN with a cell containing multiple alternative operations, such as convolutions with different kernel sizes. The selection among these alternatives is controlled by a set of architectural weights optimized during training. Higher weights are assigned to the paths that perform better according to a predefined evaluation metric. At the end of the training, a discretization step extracts a final architecture by selecting, for each module, the alternative with the highest architectural weight. These methods are commonly referred to as *path-based* DNAS.

To identify both accurate and efficient architectures, DNAS enhances the standard training loss with a differentiable regularization term encoding network complexity. Standard complexity measures include the number of parameters, the number of operations per inference (OPs), or differentiable estimates of hardware-related costs such as latency and energy consumption [35]. The overall optimization problem in DNAS is then formulated as follows:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \lambda \mathcal{R}(\theta) \tag{2.11}$$

where θ denotes the set of architectural weights encoding different paths in the supernet, W represents the trainable weights of the network and λ is a regularization coefficient that balances the trade-off between task performance and model complexity.

Despite path-based DNAS techniques being faster than black-box optimization ones, exploring large search spaces requires training a large supernet with non-negligible computational and memory overhead.

A more efficient alternative is represented by *mask-based* DNAS [33, 31]. Instead of constructing a supernet with multiple alternative paths, these methods define the

search space by applying modifications to a standard, single-path neural network, commonly denoted as the *seed* network. The search space consists of all possible architectures derived from the seed by reducing its complexity, for instance, by decreasing kernel sizes or reducing the number of channels. The optimization goal is to selectively remove unnecessary components from each layer while preserving task performance. Sub-architecture exploration is achieved by associating trainable binary *architectural masks* with the network's weights and activations. Each mask element corresponds to a specific subset of weights or activations, such as a channel or a portion of a convolutional filter. During the forward pass, the masks are applied via a Hadamard product, effectively removing the masked-out components and simulating their absence. The values of these masks are optimized continuously, as done with the architectural weights in path-based DNAS, to eliminate redundant elements while maintaining accuracy. In practice, the optimization process assigns higher mask values to important weights and activations, ensuring they are kept. In comparison, lower mask values indicate less critical components that can be pruned. At the end of the search, the final architecture is constructed by discarding all network components corresponding to mask values that round to zero.

As mentioned at the end of the previous Sec. 2.2.1, mask-based DNAS and gradient-based structured pruning can be considered equivalent under certain conditions. In particular, when structured pruning is cast as a DNAS problem, it explicitly controls key hyperparameters of the network—such as the number of channels and the kernel size. By selectively pruning along specific dimensions of the weight tensor, this approach effectively reduces these hyperparameters, e.g., by eliminating channels or reducing filter dimensions. The structured pruning techniques presented in this thesis will always be framed as mask-based DNAS algorithms.

Mask-based DNAS represents a more efficient alternative than path-based techniques, given its limited memory and computation overhead compared to standard seed training. However, mask-based DNAS supports a more constrained search space since it can only remove elements from the seed architecture; thus, it cannot explore changes in network depth or layer types. Despite this limitation, mask-based DNAS allows for finer-grained control over architectural modifications. For example, given a convolutional layer with 32 output channels, the search can consider all possible configurations from 1 to 32 channels with single-channel granularity. This level of flexibility would be impractical to achieve using a supernet with multiple paths [34].

2.2.3 Quantization and Mixed-Precision Search

The background about quantization presented in this section will be helpful to understand Ch. 4.

When working with edge devices, integer quantization is widely employed to replace the floating-point representations of both weights and activations with integers. This replacement not only reduces the memory footprint of the model but also enables the use of integer arithmetic, which is faster and more energy-efficient [27]. Furthermore, integer quantization enables the execution of DNNs on edge devices that do not include a floating-point unit. Quantization may be applied to networks trained in floating point (post-training quantization) or simulated during training (quantization-aware training), with the latter often yielding improved accuracy [27].

In this thesis, we consider the well-known *affine quantization* scheme, which maps generic floating-point tensors $\mathbf{T}$ to n-bit integers as follows:

$$\mathbf{T}_n = \underset{0:2^n-1}{\text{clamp}} \left(\text{round} \left(\frac{\mathbf{T} - \alpha_{\mathbf{T}}}{\varepsilon_{\mathbf{T}}} \right) \right) \tag{2.12}$$

Here, the clamp function restricts the values to the interval $[0, 2^n - 1]$, and the range $[\alpha_{\mathbf{T}}, \beta_{\mathbf{T}})$ defines the interval that can be mapped without saturation. The quantization step is given by $\varepsilon_{\mathbf{T}} = (\beta_{\mathbf{T}} - \alpha_{\mathbf{T}})/(2^n - 1)$. Typically, $\alpha_{\mathbf{T}}$ and $\beta_{\mathbf{T}}$ are determined independently for each *layer*, *block*, or *channel* of weights or activations. While layer-wise and channel-wise assignments are similar, block-wise quantization introduces significantly higher memory overheads. It necessitates major modifications to the layer execution flow (as each block of weights requires separate rescaling). Consequently, block-wise quantization is incompatible with most DNN accelerators or inference libraries for general-purpose hardware. For this reason, the work discussed in this thesis adopts per-layer and per-channel quantization parameters. This affine quantization scheme has been studied in works such as PACT [81] and LQ-Nets [82], which also learn the value range and the optimal quantization step during training.

Conventional quantization employs a *fixed-precision* scheme where a uniform bit-width (typically 8 bits) is applied throughout the model. Recently, however, *mixed-precision* strategies have attracted considerable interest [83]. These strategies assign different bit-widths to various sections of a DNN, creating additional

optimization opportunities regarding memory, computation time, and energy consumption. This is especially true when the hardware natively supports operations at sub-byte precision [3, 84]. Nevertheless, determining the optimal allocation of bit widths across the network is challenging due to the exponentially growing search space with an increasing network depth. Approaches inspired by NAS [36, 79] have recently been applied to address this Mixed-Precision Search (MPS) problem, and these methods, similarly to NAS, can be classified into two families: *black-box* and *one-shot*. As discussed in Sec. 2.2.2, also for the MPS problem, black-box methods leverage iterative optimization techniques such as RL [85, 86], and while flexible, they scale poorly with the size of the search space.

In contrast, one-shot MPS methods, similar to DNAS algorithms, combine the precision search and training into a single joint loop, thereby addressing the primary limitation of iterative MPS approaches. This is achieved by formulating an analytical, differentiable objective that simultaneously optimizes network weights and bit-width assignments using gradient descent. Both iterative and one-shot state-of-the-art MPS schemes will be reviewed in Ch. 4.

2.2.4 Knowledge Distillation

The last key optimization technique reviewed in this background section is Knowledge distillation (KD) [87]. Although this technique will not be used in the novel technical work presented in this thesis, is described here for completeness. Nonetheless, this method can be seen as an orthogonal approach to enhance and recover some of the task-performance loss introduced by pruning, neural architecture search, and quantization.

The general idea of KD is having a large and task-performant *teacher* network, whose logits denoted z^{t} are used to guide a smaller *student* with logits z^{s}. As an example the teacher can a large and performant network, while the student can be a smaller network derived from the teacher using pruning or it can be a deeply quantized version of a floating-point teacher.

This teacher-student relationship is described using a novel KD loss $\mathcal{L}_{KD}$ described in Eq. 2.13:

$$\mathcal{L}_{KD} = (1-\alpha)\,\mathcal{L}\big(y, \sigma(z^{\mathrm{s}})\big) + \alpha T^2\,\mathrm{KL}\Big(\sigma(z^{\mathrm{t}}/T)\,\big\|\,\sigma(z^{\mathrm{s}}/T)\Big), \tag{2.13}$$

where σ is the softmax, $T > 1$ is an optional temperature term to smooth the predictions, and $\alpha \in [0,1]$ balances the standard task-specific loss $\mathcal{L}$ with the Kullback–Leibler (KL) divergence which is a measure of similarity between the teacher and student outputs. By learning to *imitate*, the student typically attains task-performance close to the teacher while having a possibly much simpler architecture, which is essential for edge deployment and real-time applications.

2.3 Hardware Targets

2.3.1 STM32

The first target platform is the STM32H7 [1], an off-the-shelf MCU manufactured

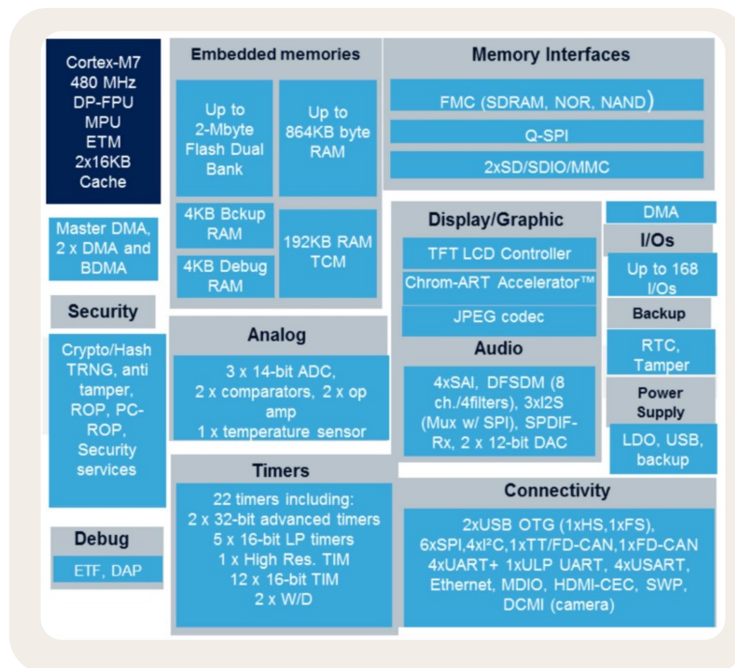

Fig. 2.2 STM32H7 MCU block diagram [1].

by ST Microelectronics. This platform will be one of the deployment targets used in Ch. 3. The MCU block diagram is shown in Fig. 2.2.

The STM32H7 includes one core ARM M7. This platform is designed to maximize the computational capability for general-purpose computing in a tight

power envelope. Moreover, it contains two caches to improve the core performance. Precisely, the cache system consists of a data and an instruction cache that allows to reduce the time to fetch instructions and move the data to the register file. The main component is the ARM Cortex M7, a general-purpose processor with floating-point computation support. On-chip SRAM and Flash memories are connected through the bus system. Additionally, Direct Memory Access (DMA) peripherals are connected to support the connection of external memories or to move data rapidly from the sensor to the core. The highest operating frequency is 480 MHz, at a power consumption of 234 mW. This MCU also supports a wide range of peripherals, including Serial Peripheral Interfaces (SPI) and Inter-Integrated Circuits (I2C) for communication or Analog-to-digital converters (ADCs) for sensor data connection.

2.3.2 GAP8

Another platform widely used in this thesis (Ch. 3, Ch. 5, and Ch. 7) as deployment

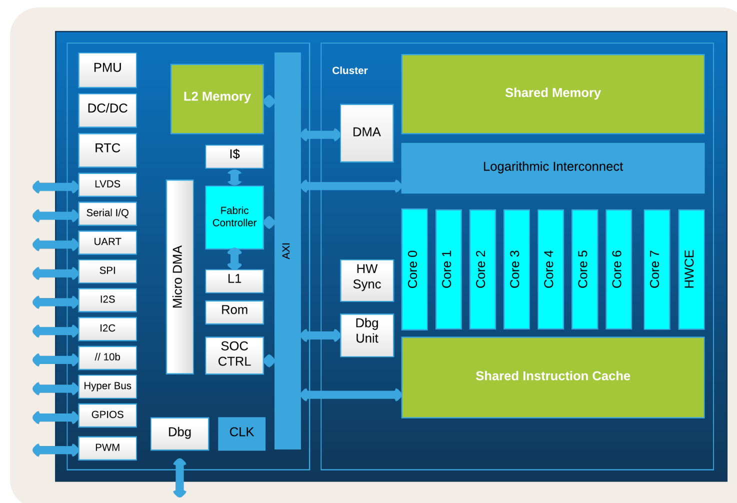

Fig. 2.3 GAP8 SoC block diagram [2].

and optimization target is the GAP8 [2] IoT multi-core SoC, manufactured by GreenWaves Technologies. The schematic block diagram of GAP8 is shown in Fig. 2.3.

The main functional units of GAP8 are the *fabric controller*, which is a single 32-bit RISC-V core, and the *cluster* which is composed of 8 RISC-V cores. The fabric controller controls the cluster and other peripherals. The octa-core cluster is used

for computationally intensive tasks, such as DNNs' inference. It supports vectorized and parallelized computations, thanks to customized instruction set extensions [88]. An internal two-level memory hierarchy characterizes GAP8. All cores can reach the 512 kB level two (L2) memory. In contrast, the level one (L1) memory is divided into two sections: 16 kB dedicated to the fabric controller and 64 kB shared memory for the cluster. If needed, a third larger but slower level three (L3) off-chip memory can be reached via a HyperBus interface or a quad-SPI. To deal with such a complex memory hierarchy, GAP8 features two DMA co-processors that can help hide L2 and L3 memory latency by moving data within the hierarchy coupled with a double-buffering strategy. In this way, simultaneously with memory transfers, the RISC-V cores perform computations on previously transferred data. Since GAP8 lacks a floating-point unit (FPU), the deployed models must be quantized to 8-bit.

2.3.3 Hardware with Mixed-Precision Support

This section introduces specialized computing units supporting mixed-precision inference. These HW platforms will be used as optimization and deployment targets in Ch. 4.

Mixed Precision Inference Core

The Mixed Precision Inference Core (MPIC) [3] is a RISC-V core based on the

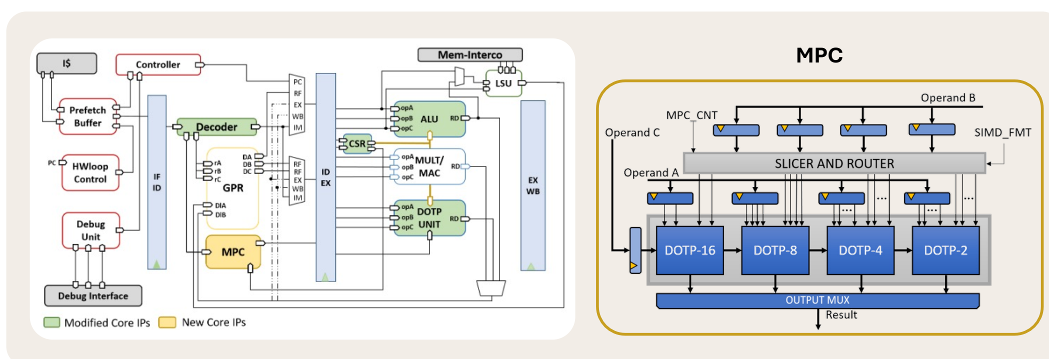

Fig. 2.4 MPIC block diagram [3].

open-source RI5CY design; It features a 4-stage pipeline and supports all standard RISC-V extensions. It also supports the XpulpV2 extension, a DSP-oriented extension with post-increment loads/stores, hardware loops, and single-instruction

multiple-data (SIMD) instructions. MPIC improves this core by extending the SIMD support down to 2 bits. Specifically, MPIC includes a dot-product unit (shown in the right part of Fig. 2.4) dedicated to parallelize multiply-accumulate (MAC) operations on independently quantized inputs (weights and activations). It can execute parallel operations such as 2×16-bit, 4×8-bit, 8×4-bit, or 16×2-bit, gaining speed from the lower bit-widths. Additionally, further speedup is achieved compared to homogeneous precision, thanks to fewer memory fetch operations.

Neural Engine 16

Neural Engine 16 (NE16) [84] is an implementation of the NEeureka accelerator [4],

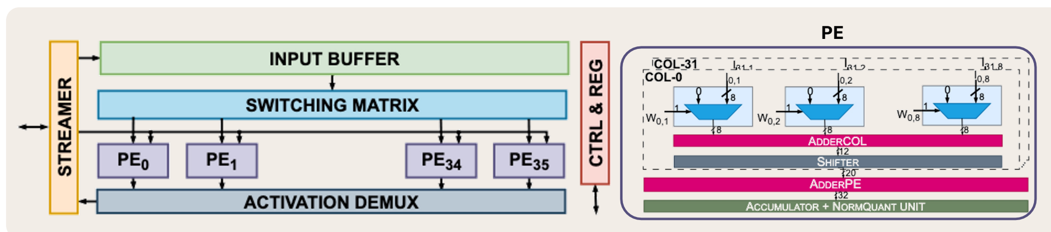

Fig. 2.5 NE16 block diagram [4].

consisting of 3×3 processing elements (PEs). Each PE (shown in the right part of Fig. 2.5) calculates a single convolution output spatial pixel across 32 output channels. This accelerator is optimized for performing 3×3 standard convolutions, 1×1 pointwise convolutions, and 3×3 depthwise convolutions. It supports weights with a precision ranging from 2 to 8 bits and 8-bit quantization for activation tensors. NE16 employs 1×8-bit parallel AND gates as basic blocks, configured differently depending on the required operating mode. Specifically, when the bit width is >1, weight bits are distributed across multiple basic blocks, and their outputs are appropriately combined. Consequently, the latency of MAC operations scales with the precision of the weights.

2.3.4 Multi-accelerator System-on-Chips

This section introduces SoCs, including multiple computing units that expose a trade-off between task performance and latency/energy consumption. These platforms will be used as deployment targets in Ch. 6.

DIANA

The DIANA architecture, detailed in [5] and shown in Fig. 2.6, is based on a

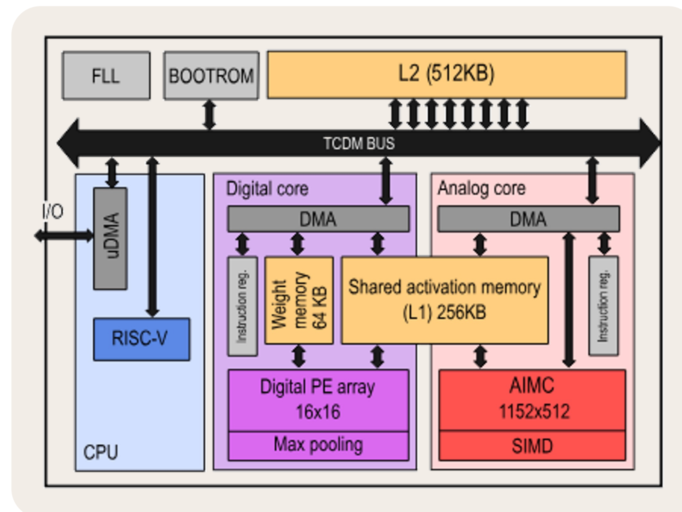

Fig. 2.6 DIANA block diagram [5].

single-core RISC-V CPU which controls two specialized DNN CUs. The first CU consists of a 16×16 grid of digital processing elements, each capable of performing MAC operations at 8-bit precision and equipped with 64 kB of weight memory. The second CU is an Analog In-Memory Computing (AIMC) unit containing 500,000 cells, utilizing ternary weights. Both CUs share a 256 kB L1 memory with dual Read/Write ports that are accessed via DMA.

Darkside

The Darkside SoC [6] is schematized in Fig. 2.7 and represents an upgrade of the GAP8 platform discussed in Sec. 2.3.2. As GAP8, it features the general-purpose fabric controller and an 8-core cluster of RISC-V processors. Additionally, it includes the Depthwise Convolution Engine (DWE), which is dedicated to accelerating depthwise convolutions. The cluster and the DWE share an L1 memory consisting of 32 4-kB banks capable of handling up to 32 parallel requests. A 256 kB L2 memory completes the hierarchy, which is accessed via a dedicated DMA co-processor.

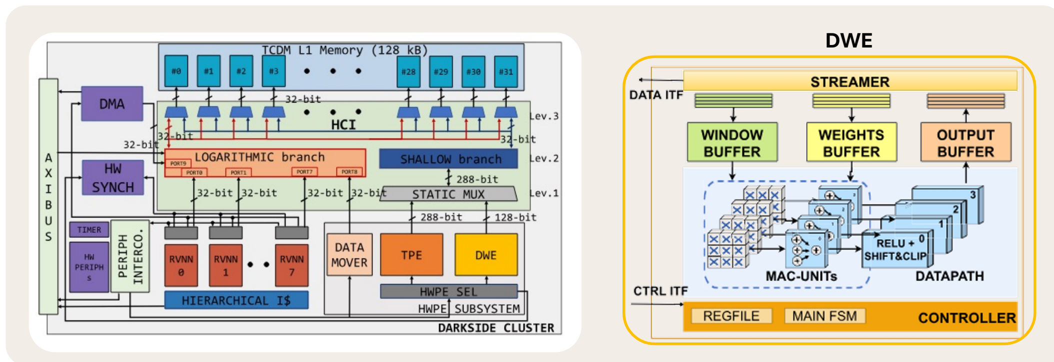

Fig. 2.7 Darkside block diagram [6].

2.3.5 HWatch Platform

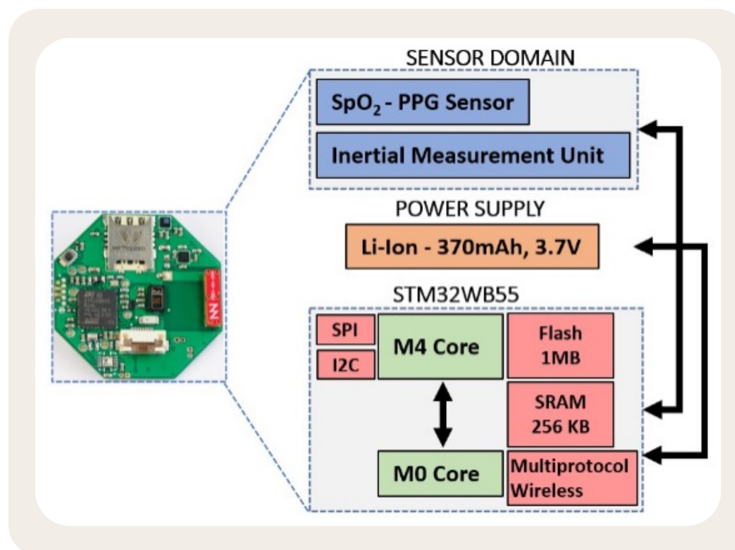

Fig. 2.8 HWatch block diagram [7].

The HWatch [7] is a custom board with the form factor of a smartwatch designed to develop wrist-worn applications. It will be used as a deployment target in Ch. 7 when targeting PPG-based heart-rate estimation. The block diagram of the system is shown in Fig. 2.8.

The watch includes an STM32WB55RGV6 SoC from ST Microelectronics [89], named STM32WB hereafter. The SoC comprises two fully independent cores: an Arm Cortex-M0+ core at 32 MHz (network processor) and an Arm Cortex-M4 core

running at 64 MHz (application processor). The STM32WB also includes a radio stack that is compliant with the Bluetooth Low Energy (BLE) 5.0 standard. The primary power source is a Li-Ion 370 mA@3.7V battery.

The watch also includes two sensors. The first is the MAX30101 [90], a low-power pulse oximeter that collects a PPG signal from which heart rate can be derived. Then, the LSM6DSM [91] 6-D inertial measurement unit is included. The LSM6DSM also features a Machine Learning (ML) core optimized for executing Random Forests (RFs). The two are connected with the MCU using I2C and SPI protocols, respectively.

2.3.6 MAUPITI

MAUPITI is a smart sensor prototype manufactured by ST Microelectronics in

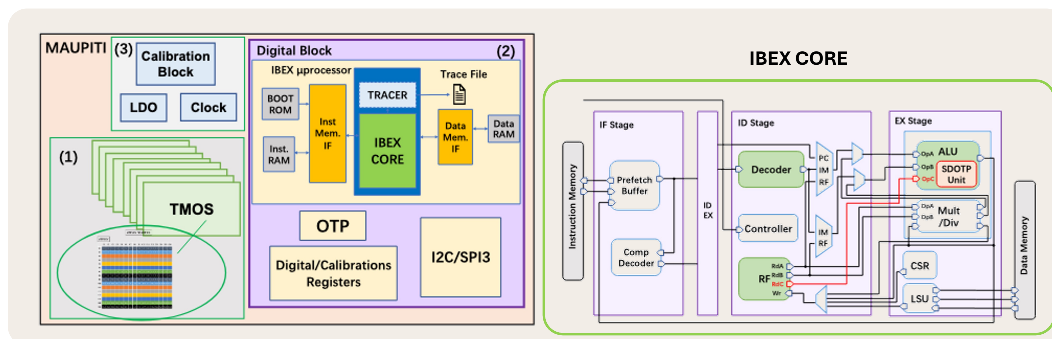

Fig. 2.9 MAUPITI block diagram [8].

collaboration with the Politecnico di Torino [8]. The sensor is clocked at 20MHz and supports thermal image acquisition with a maximum frame rate of 10 Frames Per Second (FPS).

The system is schematized in Fig. 2.9 and includes two main components. The first is the sensing part, made of a 16x16 array of thermal MOSFET (TMOS), sensitive to infrared radiation. The total consumption of the array is 0.62mW with each TMOS drawing $\approx 1\mu A@2.4V$.

The second part is devoted to the digital processing of the acquired signals. It includes a customized RISC-V IBEX [92] core with two dedicated 16kB memories for instruction and data.

The IBEX core has been customized to support quantized DNNs using `INT4` and `INT8` data formats, with the addition of an efficient arithmetic unit supporting low bit-width integer SIMD Sum of Dot Product (SDOTP) operations.

Chapter 3

Structured Pruning of Temporal Convolutional Networks

As introduced in Sec. 2.2.1, one of the key optimization techniques to enable DL inference on edge devices is represented by *structured pruning*. Indeed, starting from a given DNN, structured pruning allows the elimination of entire groups of model parameters, such as filters or channels, thus reducing the overall complexity of the input model [66]. Therefore, this complexity-reduction technique is a good candidate for meeting the constraints of edge platforms, such as limited memory and computational resources, power efficiency requirements, and real-time inference needs. This approach inherently preserves the dense structure of the model, allowing the reuse of the existing edge inference libraries without the need for developing new ad-hoc kernels, thereby reducing engineering cost and ensuring an easier deployment [66]. Structured pruning removes entire blocks of parameters aligned with architectural hyper-parameters, such as the number of channels or kernel sizes in NNs.

Despite the advantages of structured pruning, much of the existing State of the Art (SotA) focuses on specific aspects, such as reducing the number of channels or filters, while neglecting other dimensions that could be optimized [66]. This narrow focus limits the potential for broader applicability and efficiency gains, for example, specialized tasks like time series processing, which are increasingly prevalent in edge applications [46, 49]. Indeed, while RNNs [93] have historically been the SotA DL models for time-series tasks, TCNs [61] have emerged as a competitive

alternative. TCNs have demonstrated comparable accuracy to RNNs, concurrently offering computational benefits, including increased arithmetic intensity, reduced memory footprint, and enhanced data reuse [61]. For these reasons, TCNs are particularly advantageous for edge computing deployments.

In this chapter, we will present *Pruning In Time* (PIT), the first hardware-aware gradient-based structured pruning approach capable of optimizing the key hyper-parameters of convolutional and fully-connected layers in TCNs. Namely, we optimize dilation, receptive field, the number of channels for convolutional layers, and the number of output features for FC layers. As stated in Sec. 2.2.2, gradient-based structured pruning and mask-based DNAS are twin concepts. For this reason, PIT will also be presented as a search approach over the space of the possible targeted hyper-parameters for the given TCN.

This chapter first provides an overview of the related works in Sec. 3.1, highlighting SotA techniques. Then, Sec. 3.2 presents the details of the PIT pruning algorithm. Finally, Sec. 3.4 presents the experimental validation of the method by comparing it with other SotA approaches and deploying a subset of the optimized TCNs on two edge platforms.

Moreover, the PIT structured pruning algorithm will be used in Sec. 7.1 of Ch. 7 to derive TCNs with different task-performance vs complexity tradeoffs. These TCNs are then deployed in a real setup composed of a smartwatch and a smartphone where the TCNs and the two devices are used to perform collaborative Heart-rate estimation.

The work described in this chapter has been published in [94, 34].

3.1 Related Works

As mentioned in the introduction of this chapter, no existing gradient-based optimization methods directly target the search space associated with the TCNs under consideration. Consequently, in this section, we discuss related gradient-based methods that can be potentially extended to support this search space.

The first class of gradient-based approaches capable of supporting such search space is the path-based DNAS algorithm family, already introduced in Sec. 2.2.2. Nonetheless, early approaches, like DARTS [79], are limited in the number of

explorable alternatives given that the supernet needs to fit the memory available in the HW used for training. This problem is partially solved by more advanced methods like ProxylessNAS [35], a path-based DNAS that, for each batch of inputs, keeps in memory no more than two supernet paths, thus reducing the memory requirements. ProxylessNAS employs an alternated optimization scheme to train network weights and supernet architectural parameters. Initially, the architectural parameters are kept constant, and a sub-architecture is randomly sampled from the supernet based on their current values. Subsequently, the weights of the sampled sub-architecture are updated using the training dataset. Conversely, with the network weights kept constant, the architectural parameters are updated utilizing a validation set. During this phase, two distinct architectural paths are simultaneously sampled from a multinomial distribution and updated. This approach increases the search space size while keeping the memory overhead constant.

Regarding mask-based DNAS and structured pruning approaches, we have FBNetV2 [33] and MorphNet [31]. As described in the background Sec. 2.2.2, with these approaches, a large architecture with a unique path replaces the supernet. Optimized architectures are discovered as sub-architectures of this input seed model by training a set of trainable masks that actively turn off network parts. FBNetV2 [33] uses a set of dedicated masks weighted with a trainable parameter. Each mask encodes a different spatial resolution or a different number of output channels. At the end of the optimization, the mask coupled with the largest parameter is used to determine the final hyper-parameter setting. FBNetV2 [33] employs a collection of discrete masks, each representing a distinct configuration of output channel counts or spatial resolutions, and associates each mask with a trainable scalar weight. The final hyper-parameter configuration is determined by selecting the mask corresponding to the maximum learned weight. Similarly, MorphNet [31] utilizes the multiplicative scaling factors included in BN layers [59] as masking parameters. Channels from the preceding convolutional layer are pruned when these scaling factors are smaller than a predefined threshold.

The literature mentioned above predominantly addresses 2D-CNNs for CV applications. As said, these existing methodologies have not been extended to time-series processing despite the many one-dimensional time-dependent signals in edge-relevant real-world applications such as biosignals [19], audio [14], and sensor readings from anomaly detection [16]. PIT aims to bridge this gap by introducing a novel approach to optimize 1D networks.

3.2 Proposed Method

In this section, the PIT approach is introduced. PIT is a novel structured pruning approach developed during this PhD thesis that is tailored for tasks regarding classification and regression over time series. In particular, PIT explores the main hyper-parameters of TCNs. As introduced in the background Sec. 2.1.1, these key hyper-parameters are the *number of output channels* C_{out}, the *receptive-field* F, and the *dilation factor* d. Moreover, PIT can also explore the number of output features of FC layers as these can be framed as a corner case of a 1D convolutional layer with kernel size K, receptive-field F, and dilation factor d all equal to 1. To simplify the notation and the discussion, in this section, we will always refer to convolution output channels C_{out}, knowing that a similar discussion can be immediately extended for the output features of FC layers.

The rest of this section is organized as follows. Sec. 3.2.1 introduces the considered search space and the employed search logic based upon structured pruning and how this is tailored to each considered hyper-parameter (Sec. 3.2.1- 3.2.1). Then, Sec. 3.2.2 explains how the standard task-loss $\mathcal{L}$ is augmented with cost-aware regularization targeting the number of parameters (Sec. 3.12) and the number of operations (Sec. 3.2.2). Sec. 3.2.3 details the general training procedure.

3.2.1 Search-Space

Fig. 3.1 depicts the PIT search space, including all sub-architectures derived from a seed TCN by modifying the previously mentioned hyperparameters. Specifically, compared to the seed, PIT can *reduce* C_{out} or F, and *increase* d, which collectively decreases the layer's complexity and memory requirements.

To achieve this, each convolutional layer in the seed model is transformed into a function $L_n(W^{(n)};\theta^{(n)})$, defined by its original weight tensor $W^{(n)}$ and a new set of architectural parameters $\theta^{(n)}$. For a TCN with N layers, the PIT search space will be represented as:

$$\mathcal{S} = \{L_n(W^{(n)};\theta^{(n)})\}_{n=0}^{N-1} \quad (3.1)$$

During the search process, a *binary mask* $\Theta^{(n)}$ is created by combining the elements of $\theta^{(n)}$. This mask is then responsible for *pruning* specific portions of the layer's weights. Namely, at each iteration, an architecture $\hat{\mathcal{S}}$ is sampled from $\mathcal{S}$ by computing

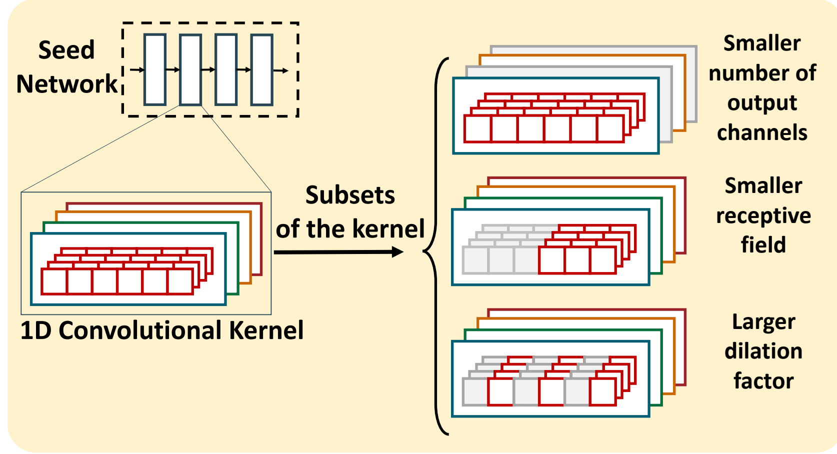

Fig. 3.1 Search space considered by PIT

$\hat{\mathcal{S}} = \{L_n(W^{(n)} \odot \Theta^{(n)})\}_{n=0}^{N-1}$, i.e., the Hadamard product between $W^{(n)}$ and $\Theta^{(n)}$. In this process, the portions of $W^{(n)}$ associated with the mask's 0-valued elements are removed, effectively allowing the seed layer to replicate the behavior of a model with fewer channels, a smaller receptive field, or increased dilation. Details on how $\Theta^{(n)}$ is derived from $\theta^{(n)}$ to achieve these modifications are presented in Sections 3.2.1-3.2.1.

Using *binary* masks is necessary to either completely remove slices of $W^{(n)}$ (by setting their value to 0) or retain them unchanged (by setting their value to 1) when sampling an architecture through the Hadamard product. This ensures that only *feasible* architectures are sampled, with integer values for C_{out}, F, and d. To achieve this, $\Theta^{(n)}$ is binarized during the forward pass of the search/training process by applying a Heaviside step function with a threshold $th = 0.5$.

Additionally, to integrate the search process into a standard gradient-based network training where both weights $W^{(n)}$ and architectural parameters $\theta^{(n)}$ are learned simultaneously, it is necessary to make the $\theta^{(n)} \to \Theta^{(n)}$ transformation differentiable. To do this, we must deal with the challenges posed by the Heaviside function's derivative, i.e., zero almost everywhere and undefined at th. This issue can be faced by following the BinaryConnect [95] approach, which employs a Straight-Through Estimator (STE), where the forward function is kept as it is while the gradient backward function is replaced with an identity.

For the sake of notation clearness, in the rest of the discussion we divide the parameters $\theta^{(n)}$ into three groups: $\alpha^{(n)}$, which control the number of channels C_{out}; $\beta^{(n)}$, used to adjust the receptive field F; and $\gamma^{(n)}$, which tune the dilation

factor. Moreover, we omit the superscript (n) when it is not required to streamline expressions. In PIT, each parameter group generates an *independent* binary mask, which can then be combined with the other two. This independence allows PIT to optimize C_{out}, F, and d individually or jointly. During a joint search, PIT can explore up to:

$$|\mathcal{S}| \approx \prod_{n=0}^{N-1} (C_{out,seed}^{(n)} \cdot F_{seed}^{(n)} \cdot \lceil \log_2(F_{seed}^{(n)}) \rceil) \quad (3.2)$$

different solutions, where $C_{out,seed}$ and F_{seed} in (3.2) refer to the seed layer's original hyper-parameters. The logarithmic term in (3.2) arises because only power-of-2 dilation factors are considered, as explained in Sec. 3.2.1. For a relatively small seed network with $N = 8$, $F_{seed}^{(n)} = 17$, and $C_{out,seed}^{(n)} = 128\ \forall n$, this results in evaluating approximately $\approx 10^{32}$ architectures during a single training run.

Channels Search

To determine the number of channels in each convolutional layer, we build upon

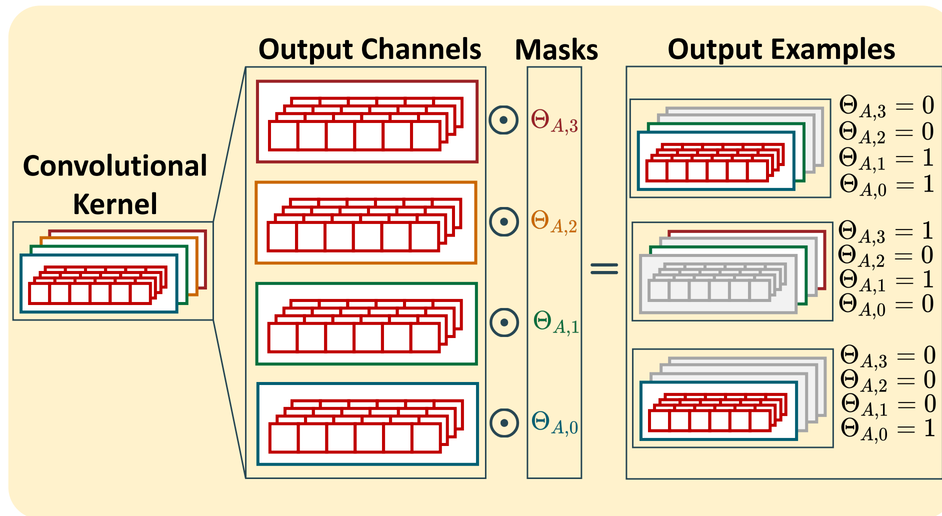

Fig. 3.2 Example of channels search. Each $\Theta_{A,m} = 0$ masks a slice of size $K \times C_{in}$ of the weights tensor W corresponding to the m-th convolutional filter.

the approach introduced in [31]. That method transforms the multiplicative trainable parameter of BN layers [59] into a binary mask, allowing entire output channels to be pruned, hence enabling exploration of sub-layers where $C_{out} < C_{out,seed}$. However, since this method requires a BN layer after every convolution, a standard practice in modern 2D-CNNs, it limits its broader applicability. PIT overcomes this limitation

by decoupling the channel search from BN layers and introducing a dedicated trainable set of parameters, α, to prune entire channels directly from the W tensor of convolutional layers. PIT evaluates each channel individually by defining an α array with a length equal to $C_{out,seed}$ and using it to create binary masks as:

$$\Theta_A = \mathcal{H}(|\alpha|) \tag{3.3}$$

where $\mathcal{H}$ denotes the Heaviside binarization function. In this way, the layer function of a single temporal convolution defined in Eq. 2.9 is updated to:

$$\tilde{y}_t^m = \sum_{i=0}^{K-1} \sum_{l=0}^{C_{in}-1} x_{ts-di}^{l} \cdot (\Theta_{A,m} \cdot W_i^{l,m}) \tag{3.4}$$

Each binarized mask element is multiplied to all weights corresponding to its associated *convolutional filter*, effectively masking an entire slice of the weight tensor along the output channels axis. Weights associated with a 0-mask are removed, hence pruning the corresponding output channel from the layer. Fig. 3.2 shows how Θ_A parameters are applied to a simple layer with $C_{out,seed} = 4$.

Moreover, PIT is not limited to reducing the number of channels. Indeed, when skip-connections are present, it can also remove *entire* layers from the network. Specifically, if all the $\Theta_{A,m}$ values of a convolutional layer are set to zero, the input bypasses the layer entirely and flows only through the skip connection, effectively decreasing the network's depth by one layer. Conversely, in the absence of skip connections, at least one output channel is always preserved to maintain the network's connectivity.

PIT's channel search strategy varies from existing mask-based DNAS approaches such as FBNetV2 [33]. PIT masks *weights tensors* directly, whereas FBNetV2 masks *output activations*. This distinction enables PIT to smoothly extend its optimization framework to other hyperparameters, such as F and d, which are much harder to manage with activation masks. PIT use *independent* binarized parameters to control each channel, as opposed to predefined masks with gradually increasing trailing 0s, as used in [33]. As a result, PIT can prune any combination of channels, not just the trailing ones, providing greater flexibility in the optimization process.

Receptive Field Search

The second key hyper-parameter PIT optimizes is the receptive field F, which

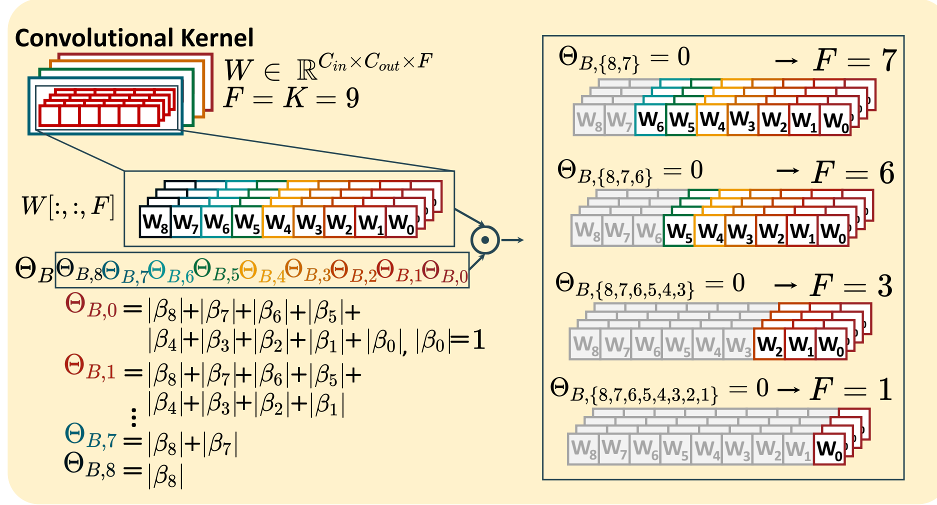

Fig. 3.3 Example of search of the receptive field. By zeroing out a time-slice of size $C_{out} \times C_{in}$ of the weights tensor W, each $\Theta_{B,i} = 0$ removes from the convolution output the contribution of 1 input time-step.

determines the range of input time steps included in a convolution operation. In standard convolutions, the receptive field equals the filter size ($F = K$). However, as explained in the background Sec. 2.1.1, this relationship no longer holds in TCNs when the dilation factor d is greater than 1. In this case, we must consider the more general formula: $F = (K-1)\cdot d + 1$. From this formula, it emerges how, by jointly optimizing F and d, PIT also indirectly modifies the filter size K.

To explore the receptive field, PIT employs an array of additional trainable parameters β of length F_{seed}. Unlike the output channels, the β parameters require further processing to define the corresponding binary differentiable masks. Simply masking *any* set of time-slices in the weight tensor does not suffice to simulate a smaller receptive field. Indeed, in causal TCN convolutions, the receptive field extends exclusively into the past. Consequently, the slices to be removed must always correspond to the “oldest" input time steps, i.e., those that are farthest back in time. To achieve this, the binary mask elements Θ_B are derived from β as follows:

$$\Theta_{B,i} = \mathcal{H}\left(\sum_{j=1}^{F_{seed}-i} |\beta_{F_{seed}-j}|\right) \tag{3.5}$$

Fig. 3.3 illustrates how $\Theta_{B,i}$ is multiplied to a time-slice of the W tensor during the forward pass. In this case, Eq. 2.9 becomes:

$$y_t^m = \sum_{i=0}^{K-1} \sum_{l=0}^{C_{in}-1} x_{ts-di}^l \cdot (\Theta_{B,di} \odot W_i^{l,m}) \tag{3.6}$$

The construction of equation (3.5) ensures that when $i > j$ then it is implied that $\Theta_{B,i} \leq \Theta_{B,j}$. This guarantees that the pruned first weight slices are always the leftmost, as shown in the example on the right side of Fig. 3.3. Furthermore, β_0 is fixed at a constant value of 1. This constraint ensures that, after binarization, $\Theta_{B,0}$ always equals 1, guaranteeing that all convolutions include no less than one time-step as input.

For the sake of efficiency, binary masks are generated using the following matrix transformation:

$$\Theta_B = \mathcal{H}\left(C_\beta \cdot |\beta|\right) \tag{3.7}$$

Here, C_β represents a constant upper triangular matrix filled with 1s with dimensions equal to each layer's F_{seed}. An example of such a matrix is depicted on the left side of Fig. 3.4.

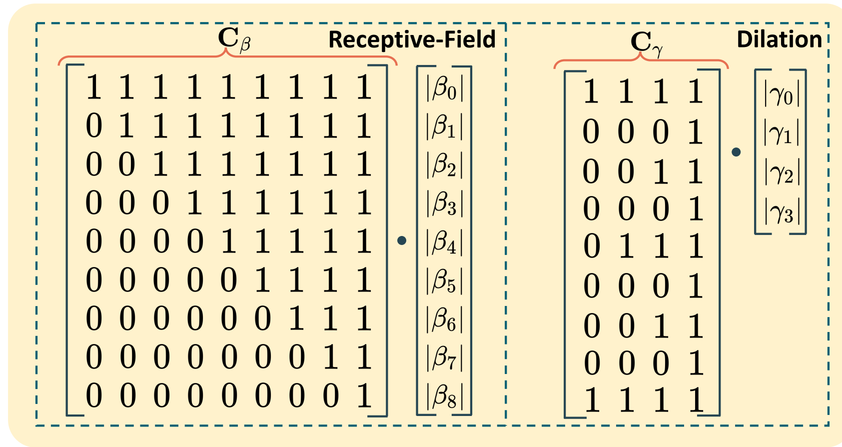

Fig. 3.4 An example of a conversion between the binary masks Θ_B and Θ_Γ and the trainable architectural parameters β and γ for a layer with $F_{seed} = 9$.

Dilation Search

Finally, the dilation factor d is also optimized using PIT. Finding the optimal dilation

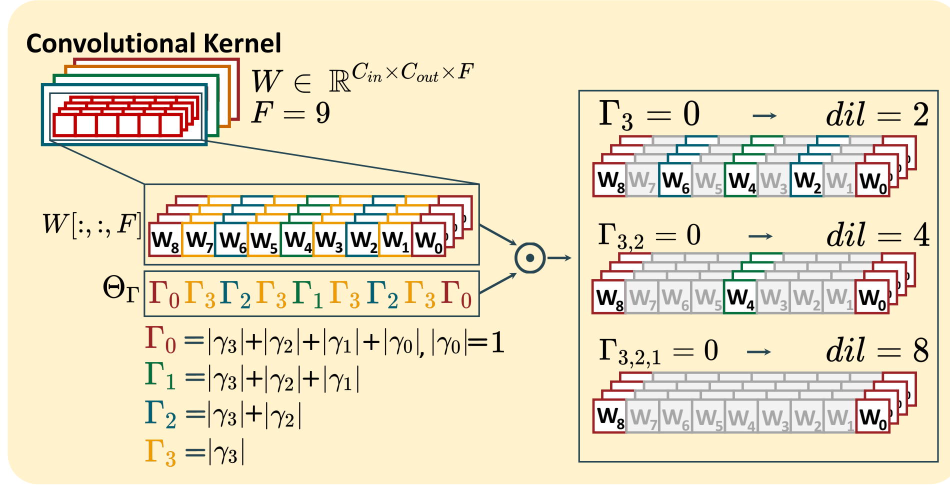

Fig. 3.5 Example of dilation search. d is increased by a factor 2 for every $\Gamma_i = 0$.

imposes specific constraints on the parts of the weights tensor that can be pruned, akin to the receptive field. In particular, it is essential to guarantee that only *regular* dilation factors are produced, which means that the intervals between successive input time steps must stay the same throughout the layer. It is unacceptable, for example, for a layer to process time steps $t, t-2, t-3$, and $t-9$ with gaps of 0, 2, and 5 time steps. Since regular dilation is necessary for low-level optimizations and efficient memory access, most inference libraries do not support irregular dilation patterns, especially those made for edge devices [96, 97].

To address these requirements, we adopt an approach similar to the one described in Sec. 3.2.1 for the receptive field. We begin with an array of trainable parameters γ, which are then combined to create differentiable binary masks. Our method supports only power-of-2 dilation factors, which are the most commonly used values and greatly simplify the mask generation process. Consequently, the length of γ is defined as $len(\gamma) = \lceil \log_2(F_{seed}) \rceil$.

The elements of Θ_Γ are derived through an intermediate array Γ, constructed similarly as done in Eq. 3.5:

$$\Gamma_i = \mathcal{H}\left(\sum_{j=1}^{len(\gamma)-i} |\gamma_{len(\gamma)-j}|\right) \tag{3.8}$$

The final mask Θ_Γ, of length F_{seed}, is generated by reorganizing the Γ_i values as follows:

$$\Theta_{\Gamma,i} = \Gamma_{k(i)}, \text{ with } k(i) = \sum_{p=1}^{len(\gamma)} 1 - \delta(i \bmod 2^p, 0) \tag{3.9}$$

where $\delta()$ denotes Kronecker's Delta function. This reorganization ensures that the Γ element with the largest index ($\Gamma_{len(\gamma)-1}$) is coupled to all positions corresponding to time steps that would be skipped when the dilation factor is $d = 2$. Similarly, the second-largest Γ element is coupled to positions skipped when $d = 4$, and so on. This approach, combined with the condition derived from Eq. 3.8, i.e., $\Gamma_i \leq \Gamma_j$ for $i > j$, guarantees that the dilation factor increases progressively. When a new Γ_i is binarized to 0, the dilation factor doubles.

The resulting Θ_Γ vector is then multiplied to the W tensor as done in Eq. 3.7, with the seed layer's dilation factor initially set to $d = 1$. To ensure the convolution is never entirely pruned, γ_0 is fixed to 1. An example of how this tensor is constructed and its effect on dilation is shown in Fig. 3.5. In practice, like the receptive field mask, Θ_Γ is derived from γ using the following matrix transformation:

$$\Theta_\Gamma = \mathcal{H}(C_\gamma \cdot |\gamma|) \tag{3.10}$$

Here, C_γ is a constant matrix consisting of 0s and 1s, which is generated procedurally based on the value of F_{seed}. An example of C_γ can be found on the right-hand side of Fig. 3.4.

Joint Search

To jointly optimize all three hyper-parameters, we apply all the corresponding Θ masks to the weight tensor of a layer. As a result, the equation for a convolutional layer during a joint search becomes:

$$y_t^m = \sum_{i=0}^{K-1} \sum_{l=0}^{C_{in}-1} x_{ts-i}^l \cdot (\Theta_{B,i} \odot \Theta_{\Gamma,i} \odot (\Theta_{A,m} \cdot W_i^{l,m})) \tag{3.11}$$

Note that, as mentioned in Sec. 3.2.1, the dilation factor for this seed layer is initialized to 1, allowing PIT to explore the entire range of possible d values. In the experiments, it is demonstrated that this joint optimization approach outperforms sequentially optimizing the three hyper-parameters. Indeed, by optimizing

them simultaneously, PIT can consider the mutual interactions between the hyper-parameters, particularly between F and d. A more detailed analysis is provided in Sec. 5.3.5.

3.2.2 Cost-aware Regularization

Following the approach of state-of-the-art DNAS methods [35, 33, 31], PIT aims to discover accurate yet low-complexity architectures by combining the task-specific loss function $\mathcal{L}$ with a regularization term $\mathcal{R}$, as defined in Eq. 2.11. This additional differentiable term introduces a prior in the loss landscape, guiding the optimization toward low-cost architectures. In this work, we consider two cost metrics: the number of parameters (or size) and the number of OPs required for a single inference. The corresponding regularizers, $\mathcal{R}_{size}$ and $\mathcal{R}_{ops}$, are expressed as differentiable functions of the *pre-binarization* masks $\tilde{\Theta}_A$, $\tilde{\Theta}_B$, and $\tilde{\Theta}_\Gamma$. These masks represent the outputs of Eq. 3.3, Eq. 3.7, and Eq. 3.10, prior to applying the Heaviside binarization. The masks themselves depend on the trainable architectural parameters α, β, and γ. We adopt pre-binarization masks, as in [31], because they provide a smoother loss landscape, improving the optimization process's convergence. The details of the two regularizers are outlined below.

Size Regularizer

The size regularizer $\mathcal{R}_{size}$ calculates the *effective* number of parameters in the network during each forward pass, using the values of the differentiable binary masks. For a convolutional layer, the number of parameters corresponding to the size of the weight tensor W is given by $C_{in} \times C_{out} \times K$. Thus, the size regularizer for a TCN with N convolutional (or fully connected) layers is defined as:

$$\mathcal{R}_{size} = \sum_{n=0}^{\text{N-1}} (\mathcal{R}_{size}^{(n)}) = \sum_{n=0}^{\text{N-1}} C_{out,eff}^{(n-1)} \cdot C_{out,eff}^{(n)} \cdot K_{eff}^{(n)} \tag{3.12}$$

where:

$$C_{out,eff}^{(n)} = \sum_{i=0}^{C_{out,seed}^{(n)}-1} \tilde{\Theta}_{A,i}^{(n)} \tag{3.13}$$

represents the effective number of channels in the n-th layer, and:

$$K_{eff}^{(n)} = \sum_{i=0}^{F_{seed}^{(n)}-1} \frac{\tilde{\Theta}_{B,i}^{(n)}}{F_{seed}-i} \cdot \frac{\tilde{\Theta}_{\Gamma,i}^{(n)}}{len(\gamma)-k(i)} \tag{3.14}$$

defines the effective kernel size, which depends on the total receptive field and the dilation factor. For the first layer in the network, $C_{out,eff}^{(n-1)}$ is constant and corresponds to the number of channels of the input signal.

The expressions in Eq. 3.13 and Eq. 3.14 provide continuous approximations of the number of *active* (non-pruned) channels and time-slices in $W^{(n)}$. Minimizing $\mathcal{R}_{size}$ encourages PIT to lower the values of $\tilde{\Theta}$ by reducing them below the binarization threshold. By adjusting the regularization strength λ in Eq. 2.11, PIT balances the trade-off between reducing costs and preserving accuracy by pruning only the $\tilde{\Theta}$ elements associated with less significant weight slices.

The denominators in Eq. 3.14 ensure that when β and γ are initialized to 1 (the default value, as discussed in Sec. 3.2.3), $K_{eff}^{(n)}$ matches the actual filter size of the seed. Indeed, each $\tilde{\Theta}_{B/\Gamma}$ is computed as the sum of a varying number of γ (or β) elements. Hence, without normalization, the estimated cost would exceed the true filter size. For example, consider a layer where $F_{seed} = 5$, and all β/γ initialized to 1. Without denominators, the resulting K_{eff} would be 33, which is incorrect. By including the denominators, K_{eff} is correctly calculated as 5, matching F_{seed}, since initializing $\gamma = 1$ implicitly sets $d = 1$.

OPs Regularizer

The second regularizer, $\mathcal{R}_{ops}$, estimates the total number of operations required for an inference. For a 1D convolutional layer, the number of OPs is given by $T \times C_{in} \times C_{out} \times K$, where T is the output sequence length as defined in Eq. 2.9. Consequently, the regularizer is expressed as:

$$\mathcal{R}_{ops} = \sum_{n=1}^{N} (\mathcal{R}_{size}^{(n)} \cdot T^{(n)}) \tag{3.15}$$

In essence, to minimize the total OPs, the regularizer incorporates the output sequence length as a weighting factor for the size cost of each layer. This distinction

becomes particularly relevant in the presence of layers like pooling or strided convolutions, which can significantly decrease T, thereby reducing the computational cost of subsequent layers in the network.

3.2.3 Training Procedure

Algorithm 1 outlines the three main phases of architecture search with PIT. The first

Algorithm 1 Gradient-Based NAS with Warmup, Search, and Fine-Tuning Phases

1: **for** $i \leftarrow 1$ to Steps_{wu} **do** # Warmup phase: optimize weights only
2: Perform gradient update on W using $\nabla_W \mathcal{L}(W)$
3: **end for**
4: **while** convergence not reached **do** # Joint search phase
5: Update both W and θ using gradient of $\mathcal{L}(W;\theta) + \lambda \mathcal{R}(\theta)$
6: **end while**
7: **for** $i \leftarrow 1$ to Steps_{ft} **do** # Final tuning phase: refine weights
8: Refine W using $\nabla_W \mathcal{L}(W)$
9: **end for**

phase involves Steps_{wu} iterations of warmup. During this phase, all θ parameters (α, β, and γ) are initialized to 1 and kept fixed. Consequently, every element of the binary masks Θ is also binarized to 1. This phase mirrors the seed network's standard training, with the sole objective of minimizing the task-specific loss function $\mathcal{L}$. The user specifies the number of warmup iterations. In our experiments, warmup is always run until convergence.

The second phase coincides with the actual search process. During this stage, the model weights W, and the architectural parameters θ are optimized simultaneously. The goal here is to minimize a combined objective: the task-specific loss $\mathcal{L}$ plus one of the regularization losses $\mathcal{R}$, as discussed in Sec. 3.2.2, weighted by the regularization strength λ. An early-stopping mechanism controls the duration of the search phase. In particular, the performance of $\mathcal{L}$ on a validation split of the target dataset is monitored, and when no improvement is observed for 20 consecutive epochs, the search is stopped.

The third and final phase involves freezing the θ parameters and their corresponding binary masks Θ to the values obtained at the end of the search phase. This step effectively samples the architecture identified as optimal during the search phase.

The selected network's weights W are then fine-tuned or retrained from scratch, using only the task-specific loss $\mathcal{L}$.

To generate multiple points in the accuracy versus cost (size or OPs) space, Algorithm 1 can be repeated with different values of the regularization strength λ. Notably, the warmup phase must only be performed once, as the seed network's final weights can be reused. Overall, Algorithm 1 has a computational and time complexity comparable to training a single TCN. Furthermore, its GPU time and memory requirements are significantly lower than those of supernet-based DNAS approaches. As a result, generating dozens of Pareto points by varying λ remains computationally feasible, as demonstrated by the timing results in Fig. 3.10.

3.3 Benchmarks

We evaluate PIT on four real-world benchmarks relevant to the edge computing scenario. These benchmarks are chosen to test the proposed algorithm's effectiveness comprehensively. Specifically, we address both regression and classification tasks. The employed seed TCNs are based on different architectural schemes, and the inputs fed to them include raw and extracted features.

3.3.1 PPG-based Heart-Rate Monitoring

The first benchmark considers Heart-Rate (HR) monitoring on wrist-worn devices, where Photoplethysmography (PPG) sensors combined with tri-axial accelerometers help mitigate motion artifacts [43, 60]. With this aim, the PPG-Dalia [43] dataset is used. The task is formulated as a *regression* of the HR value, whose ground truth is obtained from the electrocardiogram (ECG) measurements also in the dataset. All reported results adhere to the same input windowing approach and cross-validation protocol described in [43].

For this benchmark, TEMPONet is adopted as the seed network. Originally introduced in [46] and later employed for HR monitoring with SotA performance in [60], TEMPONet is a TCN comprising three *feature extraction* blocks and a *regressor* module with three FC layers. Each feature extraction block includes three convolutional layers with BN and ReLU activation, followed by average pooling.

The regressor section similarly applies BN and ReLU to its FC layers and integrates a dropout layer with a 50% rate. Compared to the original version of TEMPONet, our seed model doubles the receptive field of every convolution and sets the dilation to 1.

3.3.2 ECG-based Arrhythmia Detection

The second benchmark addresses ECG-based arrhythmia detection for wearable medical devices, using the ECG5000 dataset [98]. The task involves classifying ECG signals into five categories: Normal, R-on-T Premature Ventricular Contraction, Premature Ventricular Contraction, Supraventricular Premature or Ectopic beat, and Unclassified Beat. The reference TCN is ECGTCN, introduced in [47], which differs from TEMPONet by relying on residual blocks. The architecture begins with a convolutional layer that increases the input channel dimension, followed by three modular blocks. Each block contains two dilated convolutions with ReLU activation, BN, and a 50% dropout rate; the input and output feature maps are then summed. When input and output channels do not match, a point-wise convolution ($K = 1$) is included in the residual path to align tensor sizes. The PIT seed is derived from ECGTCN by preserving the original number of output channels and the original receptive field while setting the dilation factor of all layers to 1.

sEMG-based Hand-Gesture Recognition

The third benchmark concerns hand-gesture recognition using surface electromyography (sEMG) signals. The NinaPro DB1 dataset [44] is adopted, containing recordings from 27 healthy subjects monitored with 10 electrodes as they perform 52 distinct hand gestures, including basic finger and wrist movements, various hand poses, and grasping. The same data pre-processing and augmentation strategy proposed in [48] is employed.

The seed network is TCCNet, originally introduced in [48]. Its architecture incorporates three feature extraction blocks, each featuring two dilated convolutions with ReLU activation and a 5% dropout rate, along with a residual branch applying a point-wise convolution. The classifier includes an attention layer, as described in [99], and a final fully connected layer with 53 output neurons (52 gestures plus

one unknown class). The PIT seed is obtained by setting in all layers in TCCNet the dilation to 1.

3.3.3 Keyword Spotting

The final benchmark addresses the keyword spotting (KWS) task, using the Speech Commands v2 dataset [100], which includes 105,829 utterances from 2,618 speakers. The pre-processing procedure follows the MLPerf Tiny benchmark suite [52], resulting in 12 possible labels: 10 words plus two classes for “unknown" and “silence". The TCN employed as seed is TC-ResNet14 [49], which differs from other reference TCNs by not using dilation. Its modular convolutional blocks alternate between plain and strided convolutions (stride = 2). The receptive field in each layer is doubled to obtain the PIT seed.

3.4 Experimental Results

This section presents, discusses, and validates the results achieved by PIT across the four benchmarks discussed in the previous Sec. 3.3. Specifically, Sec. 3.4.1 presents the achieved results with the PIT search algorithm, focusing on task performance versus the number of parameters and task performance versus the number of OPs. Sec. 3.4.2 contains ablation studies conducted on the PPG-DaLIA benchmark, while Sec. 3.4.3 compares our method against a SotA path-based DNAS, ProxylessNAS [35], and two state-of-the-art mask-based DNAS techniques: MorphNet [31] and FBNetV2 [33]. Since the code for [33] is not publicly accessible, we re-implemented it based on details provided in the original paper. Lastly, Sec. 3.4.4 reports the memory, latency, and energy consumption metrics for the deployment of some networks discovered by PIT on two commercial edge devices.

PIT is implemented in Python and is based on PyTorch. All PIT searches and training experiments were executed on a single NVIDIA TITAN Xp GPU with 12GB of memory. The two deployment platforms considered are: (i) the multicore GAP-8 IoT processor by GreenWaves Technologies [2] and (ii) the single-core STM32H7 MCU by STMicroelectronics [1]. These platforms are briefly described in Sec. 2.3.2 and Sec. 2.3.1. For inference, we utilized the open-source layers library from [101] alongside the tiling tool of [102] for GAP8 and the CMSIS-NN library [103] for the

STM32H7. All deployed networks were quantized to 8 bits using PyTorch's built-in quantization framework.

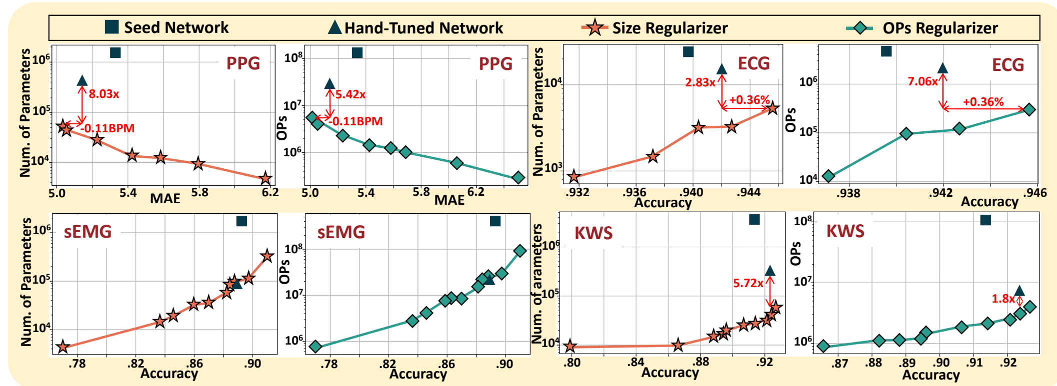

Fig. 3.6 Comparison between seed, hand-tuned TCNs, and PIT Pareto fronts for the four target benchmarks.

3.4.1 Search Space Exploration

Fig. 3.6 shows the results of applying PIT across the four benchmarks. The graphs display TCN task-performance metrics, with accuracy for classification tasks or Mean Absolute Error (MAE) for regression tasks on the x-axis and the number of parameters or OPs per inference on the y-axis. Each curve represents PIT results, where different points are obtained by varying the regularization strength (λ) and applying size and OPs regularizers. Additionally, the plots include metrics for two reference TCNs: black triangles denote the SotA, hand-tuned TCNs taken directly from prior works [60, 47–49]. In contrast, black squares indicate PIT seeds, i.e., modified versions of these networks (as detailed in Sec. 3.3) designed to expand the PIT search space.

The results for the PPG-DaLia dataset focused on PPG-based heart rate monitoring are presented in the upper-left corner. As the only regression task in the study, task performance is evaluated using MAE, with lower values being better. Starting from a single seed network, PIT discovers a diverse set of Pareto-optimal architectures, spanning an order of magnitude in parameters (4.7k–78k) and OPs (0.27M–9.6M). These PIT architectures consistently outperform the seed and the state-of-the-art TEMPONet in Pareto terms. PIT achieves a comparable MAE to the seed TCN (5.38 vs. 5.40 BPM) while using 120× fewer parameters and 96×

fewer OPs. Furthermore, PIT identifies a new SotA model, achieving an MAE of 5.03 BPM with only 53k parameters and 5.1M OPs, i.e., an improvement over the previous best model [60], which required 8.03× more parameters and 5.42× more OPs.

The upper-right quadrant displays results for arrhythmia detection using the ECG5000 dataset. PIT-generated architectures span nearly an order of magnitude in parameters (0.91k–5.36k) and OPs (50.3k–293.5k). Both the seed network and the hand-tuned alternative are Pareto-dominated. The best architecture discovered improves accuracy by +1.03% while reducing parameters by 64.7% and FLOPs by 85.8%.

The lower-left quadrant presents the results for the NinaPro-DB1 dataset, used for sEMG-based hand-gesture recognition. The architectures found by PIT exhibit similar diversity in size and OPs as in previous benchmarks. While PIT architectures outperform the seed, the hand-tuned TCNNet lies on the Pareto front. The closest PIT model achieves slightly lower accuracy (-0.47%) but reduces parameters by 3.33%. This outcome, on the one hand, highlights the quality of the original TCNNet [48]. On the other, it demonstrates the capability of PIT to generate architectures that, despite starting from an oversized seed, approach expert-designed models in quality.

Lastly, the lower-right quadrant shows results for keyword spotting on the Google Speech Commands dataset. Once again, PIT significantly outperforms both the seed and hand-tuned TCNs. The most accurate PIT model improves accuracy by +0.36% while drastically reducing parameters (-82.53%) and OPs (-44.53%). PIT architectures span 10k–98k parameters and 0.87M–3.98M OPs. The poor performance of the seeds (black squares) across all benchmarks is attributed to overfitting, caused by their excessive number of channels, large receptive fields, and lack of dilation. Table 3.1 provides the regularization strengths λ used across the

Table 3.1 Range of regularizer strength (λ) values for the four benchmarks.

Regularizer	PPG	sEMG	ECG	KWS
$\mathcal{R}_{size}$	1e-7 : 5e-4	1e-7 : 5e-6	5e-7 : 7.5e-3	5e-10 : 1e-5
$\mathcal{R}_{ops}$	1e-8 : 5e-5	5e-10 : 5e-8	5e-8 : 5e-4	1e-10 : 1e-6

four benchmarks to achieve the reported results. To ensure a balanced optimization, λ should be set so that the two components of the loss function—$\mathcal{L}$ (task-performance

term) and $\lambda\mathcal{R}$ (regularization cost-aware term)—have comparable values at the start of training. This ensures that PIT considers both task performance and inference cost during the search process, avoiding extremes where optimization focuses solely on one of the two. The specific values of λ differ depending on the task, as shown in the table.

A valuable heuristic for setting the regularization strength is to begin with a value approximately equal to $\lambda = \frac{1}{\text{Seed Model Size}}$. This is a starting point to identify the appropriate magnitude for the regularization strength. After conducting an initial PIT search with this value, adjustments can be made to increase λ to produce smaller TCNs or decrease λ to achieve ones with better task performance. Additionally, monitoring the loss during the initial epochs makes it straightforward to detect if the optimization is overly skewed toward one term. If this happens, the search can be stopped early to save training time.

3.4.2 Ablation Studies

This section studies the influence of some of the most significant PIT parameters on the PPG-based HR monitoring benchmark.

Hyper-parameters

Fig. 3.7 studies the impact of various hyper-parameters on the quality of solutions discovered by PIT. For this analysis, we use the $\mathcal{R}_{\text{size}}$ regularizer and evaluate the results in the MAE versus number of parameters space. The search is repeated three times. In each run, two of the three architectural parameter sets (α, β, and γ) are fixed to 1, allowing PIT to optimize only the third set. This setup results in the following scenarios: (i) A search that optimizes only the number of channels in each layer (*Ch-Only*), performed on a TCN with the largest receptive field and $d = 1$. (ii) A search that optimizes only the receptive field (*Rf-Only*), performed on a TCN with all channels C_{out} and $d = 1$. (iii) A search that optimizes only the dilation factors (*Dil-Only*), conducted on a network with the largest F and all C_{out} channels. The Pareto fronts obtained under these three conditions by varying the regularization strength λ are shown in Fig. 3.7. Additionally, the output of a comprehensive search that simultaneously optimizes all three hyper-parameters (*All-in-One*), i.e., the one discussed in previous Sec. 3.4.1 is included for comparison. The results indicate

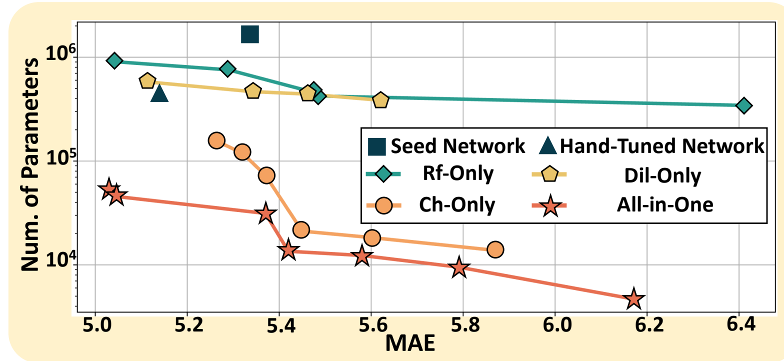

Fig. 3.7 Comparison of PIT search results with different search space definitions for PPG-DaLia.

that the primary factor driving parameter reduction is the search along the channel dimension. This is likely because the number of channels in hand-tuned TCNs often represents a significant source of redundancy, as they are typically set using general heuristics rather than being tailored to the specific task (e.g., C_{out} being a multiple of 32 or progressively increasing along the depth of the network). However, Fig. 3.7 also demonstrates that optimizing *only* the number of channels is sub-optimal. A combined optimization with receptive field and dilation hyper-parameters can discover Pareto-optimal networks spanning the entire MAE versus parameters space.

Regularizers

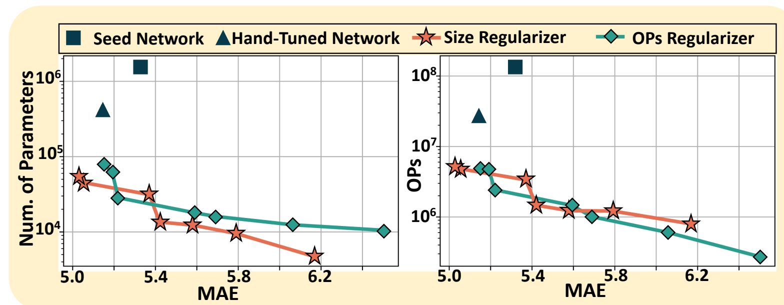

Fig. 3.8 Comparison of $\mathcal{R}_{size}$ and $\mathcal{R}_{ops}$ regularizers for PPG-DaLia.

Fig. 3.8 compares the Pareto fronts obtained using the $\mathcal{R}$size regularizer (represented by orange stars) and the $\mathcal{R}$ops regularizer (represented by green diamonds). This comparison is particularly relevant for the PPG-based HR monitoring benchmark, highlighting the distinction between model size and the number of OPs. This

distinction arises due to multiple layers, such as average pooling and strided convolution, which affect the activation array length T.

This figure shows the expected behavior with most Pareto points in the MAE versus number of parameters plane that are generated when using the $\mathcal{R}$size regularizer. The few exceptions are attributable to local minima. Conversely, the $\mathcal{R}$ops regularizer yields better or comparable solutions in the MAE versus number of OPs plane, demonstrating its effectiveness in optimizing computational complexity.

3.4.3 Comparison with SotA tools

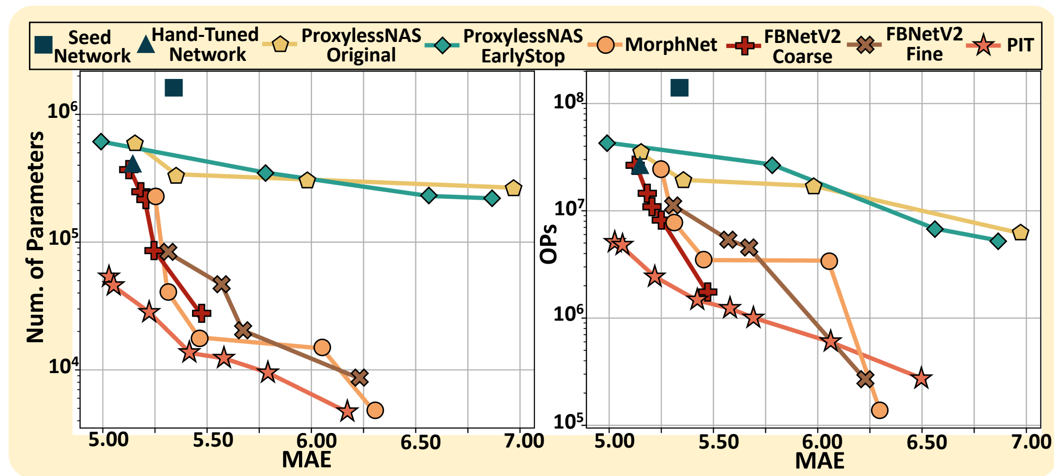

Fig. 3.9 Comparison in the Num. of Parameters vs MAE between PIT and SotA NAS and pruning tools on PPG-DaLia.

Fig. 3.9 compares the Pareto fronts obtained using PIT with those obtained using three SotA NAS and pruning tools: ProxylessNAS [35], MorphNet [31], and FBNetV2 [33], again on the PPG-based HR monitoring task. The results demonstrate that PIT outperforms all three methods across the entire design space, except for one Pareto-optimal point from MorphNet and one from FBNetV2. These points achieve a very low number of operations but at the cost of significantly higher MAE. The superior performance of PIT can be attributed to its ability to explore a larger and finer-grained search space compared to the baselines. For MorphNet and FBNetV2, this limitation is partly due to their inherent design, which does not support the exploration of the receptive field (F) or dilation (d) [31, 33]. Consequently, in their respective seeds, F and d are fixed to the hand-tuned values of the state-of-the-art

network. This difference in the initial search setup explains why, in the low-size/high-MAE regime, these tools are limited to identifying a single Pareto-optimal point.

For FBNetV2, we considered two search-space configurations: a *Coarse* space, consisting of four alternatives per layer for C_{out}, uniformly spaced (i.e., $\frac{1}{4}C_{\text{out,seed}}$, $\frac{1}{2}C_{\text{out,seed}}$, $\frac{3}{4}C_{\text{out,seed}}$, and $C_{\text{out,seed}}$), and a *Fine* space, evaluating all possible C_{out} values with a granularity of 1. While the Fine space is conceptually closer to the PIT one, the Coarse space usually achieves superior results. This is due to FBNetV2's use of a set of pre-defined binary masks with an increasing number for each layer variant, combined through Gumbel softmax. Experimentally, we observed that when the number of masks becomes too large, the search becomes unstable, leading to sub-optimal results. In contrast, PIT avoids this limitation by employing *independent* trainable masks, which operate at the channel level to keep or remove individual channels.

ProxylessNAS, being a path-based NAS, could theoretically explore the entire PIT search space if all possible versions of layers to be evaluated were included in the supernet [35]. However, this approach would result in an excessively large network, making training infeasible due to memory and computational constraints. As explained in Sec. 3.2.1, PIT evaluates C_{out} and F with a granularity of 1 and considers all power-of-2 values for d. To achieve equivalent coverage, each supernet node would require $C_{\text{out,seed}} \cdot F_{\text{seed}} \cdot \lceil \log_2(F_{\text{seed}}) \rceil$ layers connected in parallel. The parameters used in the example described at the end of Sec. 3.2.1 translate to approximately 10,000 versions of *each layer*, making such an approach infeasible. To ensure a fair comparison with PIT while maintaining a search-space size similar to that in the original ProxylessNAS paper [35], we select a coarser-grain search space for ProxylessNAS. The procedure for defining this search space is as follows. First, we conduct multiple ProxylessNAS searches independently on C_{out}, F, and d, keeping the two non-optimized hyper-parameters fixed at their seed values. Four-layer variants are included in each super-net node during each search, uniformly sampled from the PIT search space (following the same procedure described earlier for FBNetV2-Coarse). For instance, when optimizing C_{out}, each layer is configured with $\frac{1}{4}C_{\text{out,seed}}$, $\frac{1}{2}C_{\text{out,seed}}$, $\frac{3}{4}C_{\text{out,seed}}$, and $C_{\text{out,seed}}$ output channels, while maintaining the seed values for receptive field and dilation. We run ProxylessNAS multiple times for each case, varying the regularization strengths. After completing these searches for C_{out}, F, and d, we identify, for each layer, the two most frequently selected values of each hyper-parameter. The 2^3 combinations of these values are then used

to define the *combined* search space for ProxylessNAS, resulting in eight layer variants per super-net node. The two Pareto fronts shown in Fig. 3.9 are derived by running ProxylessNAS multiple times on this combined search space, using different regularization strengths. These two fronts refer to two different training schemes: the one proposed in the original ProxylessNAS paper (*Original* curve), which runs for a fixed number of epochs, and the early-stop mechanism employed for PIT (*EarlyStop* curve). As illustrated in the figure, the quality of the results is comparable between these two training schemes.

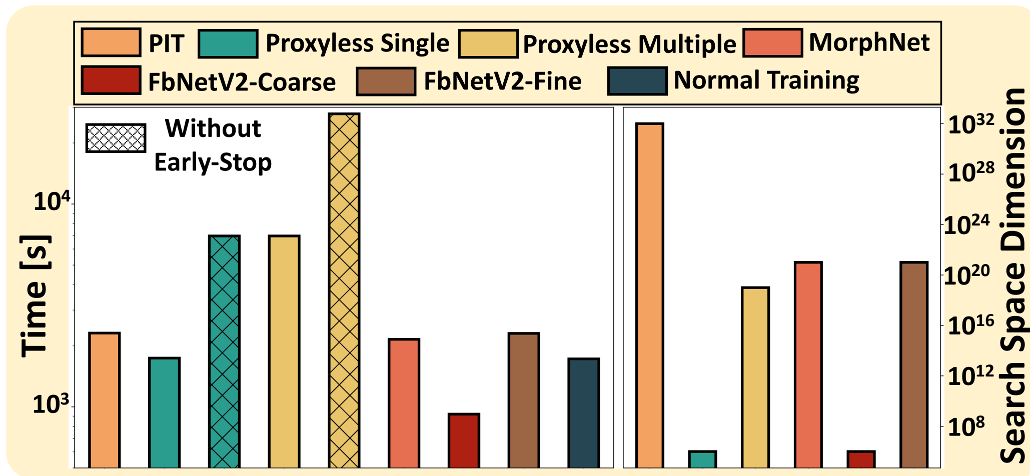

Fig. 3.10 Comparison of search time and search space dimension between PIT and state-of-the-art NAS tools on PPG-DaLia.

Fig. 3.10 compares the search-space size and average execution time for PIT, MorphNet, and ProxylessNAS on the HR monitoring benchmark. For additional context, the execution time of a standard training run for the seed network is also included. All timing measurements are limited to the search phase (excluding any warm-up) and are obtained using a single GPU with a batch size of 128. For ProxylessNAS, we show results for the initial single-parameter searches (*Proxyless-Single*) as well as the final combined search (*Proxyless-Multiple*), with both early-stopping and full training.

PIT explores a search space $10^{26}\times$ larger than Proxyless-Single and $10^{12}\times$ larger than Proxyless-Multiple. Despite this huge difference, PIT is only 1.13× slower than Proxyless-Single with early-stopping and 3.55× faster than the same method without early-stopping. When comparing against Proxyless-Multiple, PIT achieves a speedup of 3.0× with early-stopping and 14.22× without it. Compared

to MorphNet, PIT explores a search space $10^{11}\times$ larger while requiring just 1.07× more runtime. FBNetV2-Coarse completes its search in significantly fewer epochs, achieving a 2.5× speedup over PIT. However, this speed advantage comes at the expense of a much smaller search space, which is 10^{26} times smaller than PIT's. On the other hand, FBNetV2-Fine matches PIT's search time while operating on a search space 10^{11} smaller. Finally, PIT demonstrates only a modest overhead compared to standard seed network training, with an increase in runtime of just 34%.

3.4.4 Embedded Deployment

This section evaluates the deployment results of two TCN models per benchmark

Table 3.2 Deployment results on GAP8 and STM32 for the four considered benchmarks.

				STM32		GAP8	
Task	TCN	Perf. int8 (float32)	Mem. [kB]	Lat. [ms]	En. [mJ]	Lat. [ms]	En. [mJ]
PPG	HT	5.01 (5.14) BPM	423	58.3	13.6	23.2	1.2
	S	5.71 (6.17) BPM	4.7	3.2	0.75	1.18	0.06
	L	5.01 (5.03) BPM	53.2	15.2	3.56	4.25	0.22
ECG	HT	94.2 (94.2) %	15.2	6.66	1.56	2.69	0.14
	S	92.84 (93.16) %	0.9	1.8	0.42	0.78	0.04
	L	94.13 (94.13) %	5.4	2.84	0.66	1.26	0.06
sEMG	HT	88.89 (88.87) %	88.8	291	68.1	61.0	3.11
	S	86.97 (86.98) %	35.4	169	39.5	39.6	2.02
	L	91.2 (90.99) %	317.8	960	225	238	12.1
KWS	HT	92 (92.31) %	323.4	30.7	7.17	13.4	0.68
	S	87 (86.58) %	9.8	2.66	0.62	1.40	0.07
	L	92.16 (92.64) %	56.5	10.6	2.48	3.74	0.19

on the GAP8 IoT processor and the STM32H7 MCU. For each task, we deploy the top-performing model in terms of MAE or accuracy (referred to as the *Large* or *L* model). Additionally, we select a *Small* (*S*) model, which exhibits an MAE increase of less than 1 BPM or an accuracy drop of less than 5% compared to the *L* model. The hand-tuned baseline architectures (*HT*) are also deployed for comparison purposes. Table 3.2 presents the results, including performance (MAE or accuracy), memory footprint, inference latency, and energy consumption.

Despite their large difference in complexity, PIT consistently discovers competitive solutions across both hardware platforms and all four tasks. Indeed, as demonstrated in Table 3.2, the results span more than two orders of magnitude in

memory usage, latency, and energy consumption. For the PPG-based HR monitoring task, the *L*/*S* models incur MAE increases of 0/0.70 BPM compared to the hand-tuned TEMPONet but achieve significant resource savings: an 8.03/90.8× reduction in memory, and 5.45/19.6× lower latency and energy consumption on GAP8. On the STM32H7, these reductions are 3.83/18.2× for latency and energy. In ECG classification, PIT's *L*/*S* models achieve +0.07%/-1.36% accuracy relative to the hand-tuned ECGNet. They reduce memory usage by 2.83/16.8×, and latency and energy by 2.13/3.44× on GAP8 and 2.34/3.7× on STM32H7. For the sEMG-based gesture recognition task, PIT's *L*/*S* models achieve +2.31%/-1.92% accuracy compared to hand-tuned TCCNet. The higher accuracy of the *L* model comes at a cost: a 3.57× increase in memory and a 3.85× increase in latency and energy on GAP8 (3.33× on STM32H7). Conversely, the *S* model achieves a 2.51× reduction in memory, and 1.54×/1.72× lower latency and energy on GAP8 and STM32H7, respectively. These results highlight the effectiveness of the hand-tuned TCCNet for this task while showcasing PIT's ability to deliver optimized trade-offs. Finally, for the KWS task, PIT's *L*/*S* models achieve +0.16%/-5% accuracy compared to TCResNet-14. These models demonstrate substantial efficiency improvements, with memory reductions of 5.72/33.1×, and latency and energy reductions of 3.58/9.54× on GAP8 and 2.9/11.54× on STM32H7.

Chapter 4

Joint Pruning and Channel-wise Mixed-Precision Quantization

As introduced in Sec. 2.2.3, another key optimization technique is *quantization* and its extension to *mixed-precision search*. Typically, structured pruning[66] and quantization[27] are considered orthogonal optimizations acting on different redundancies. As extensively discussed in the previous Ch. 3, pruning eliminates redundant computations from a network, reducing the number of parameters and operations. Conversely, quantization and MPS optimize the data representation in a DNN, i.e., reducing redundancies in the number of bits used to represent the model's parameters and activations. MPS and pruning are usually applied alternatively or independently (e.g., one after the other), and most of the approaches that consider them jointly use time-consuming black-box methods based on RL [104] and EA [105].

This chapter presents a novel lightweight hardware-aware gradient-based method inspired by DNAS that combines pruning with *channel-wise* MPS in a holistic framework where the two optimizations happen simultaneously. This new approach avoids the lengthy process of sequential optimizations, which can unnecessarily restrict the search space and limit the options available for the second method (e.g., MPS) based on the results of the first (e.g., pruning).

This chapter first provides an overview of the related works in Sec. 4.1, highlighting SotA techniques. Then, Sec. 4.2 details the proposed optimization method. Finally, Sec. 4.3 presents the experimental validation of the method.

Moreover, mixed-precision quantization will be used in Sec. 7.3 of Ch. 7 to quantize to different precisions weight and activations of different layers of a CNN designed and trained to count people in low-resolution infrared images. This network is then deployed on a smart infrared camera with the hardware support to execute such mixed-precision networks.

The work described in this chapter has been published in [106, 107].

4.1 Related Works

4.1.1 Mixed-Precision Search

Recent studies have focused on methods to assign varying precision levels to different parts of DNNs automatically. Early attempts to address the MPS problem considered as the optimization goal only task performance. For instance, HAWQ-V2 [108] employs a sensitivity-based method based on second-order Hessian information. Lin *et al.* [109] developed a technique that optimizes the signal-to-quantization-noise ratio to determine the most effective bit-widths for each network layer. Additionally, HAQ [85] and ReLeQ [86] utilize an approach in which an RL agent assigns bit widths for individual layers. These methods reward the agent based on task performance and the hardware target's latency or energy consumption metrics. However, they often experience long convergence times due to the iterative search strategy.

Also, for MPS, one-shot algorithms represent a more lightweight solution than optimization techniques, which are black-box and non-differentiable. The first methods were based on a supernet scheme. The supernet contains all possible bit-width assignments as alternative paths in this case. The optimization goal becomes selecting a single path from the supernet, which is solved using gradient descent as in [79]. Wu *et al.* [110] propose a supernet-based approach, where the optimization is performed considering a memory cost penalty and the task-specific loss. Similarly, Gong *et al.* [111] explores precisions on MobileNet-V2 architectures using the energy consumption on the BitFusion accelerator as cost.

However, supernet-based MPS is strongly limited by the linear scaling with the search space size of computational complexity and memory. EdMIPS [83] addresses this challenge by substituting the costly parallel convolutions typical of supernet-

based methods with a more efficient unified convolution. During each iteration of the search, a single instance of the floating-point tensors is dynamically fake-quantized at various precision levels (e.g., 2-bit, 4-bit, and 8-bit), enabling exploration of multiple quantization configurations without redundant computation.

4.1.2 Joint Quantization and Pruning

The first work dealing with the combined application of pruning and quantization is Deep Compression [70]. This methodology employs a sequential implementation, initially performing pruning followed by quantization. While effective, the separate application of these techniques may neglect potential interactions between them. Furthermore, Deep Compression utilizes a uniform bit-width across the entire network. Thus, it does not consider MPS strategies.

APQ [105] integrates layer selection, structured pruning, and MPS within a unified framework. This is achieved through training a once-for-all [112] network and an ancillary neural network designed to forecast the quantized accuracy of sampled sub-networks. The search process is conducted using EA, with MPS applied layer-by-layer. AutoQ [104] utilizes an RL approach to determine the bit-width selection for each kernel within a layer, incorporating the option of kernel elimination through a 0-bit assignment. However, APQ and AutoQ are iterative, black-box methodologies requiring long search times.

Gong *et al.* [111] introduce a supernet-based, differentiable methodology for simultaneously quantizing and pruning bottleneck layers within MobileNet-V2. Specifically, a supernet is constructed wherein alternative configurations represent varying combinations of precision and pruning ratios. However, the explored search space exhibits coarseness due to per-layer quantization, and pruning is restricted to the selection of bottleneck layer expansion factors.

DJPQ [113] presents a joint pruning and quantization framework, employing non-linear quantization and Gaussian gates to modulate the weight distribution of each layer's channels. To minimize the mutual information between successive layer outputs, A variational posterior is learned. Bayesian-bits [114] proposes a framework for mixed-precision quantization and structured channel pruning, where pruning is associated with "0-bit" precision. A learnable stochastic gate is associated with each precision level, which is then trained to optimize an objective function

that balances task performance and network complexity, quantified as bit operations (bitops). A limitation of this approach is its structural constraint to explore only power-of-two precisions, which may be disadvantageous for hardware platforms such as NE16, supporting arbitrary bit-widths from 2 to 8. Furthermore, both DJPQ [113] and Bayesian Bits [114] implement per-layer mixed-precision schemes and rely on cost metrics that exhibit poor correlation with actual hardware metrics, such as bitops [115].

Chitty-Venkata *et al.* [116] propose a gradient-descent-based joint MPS and pruning technique. However, quantization is performed exclusively layer-wise, and pruning utilizes a 2:4 block-balanced strategy with a fixed 50% sparsity. This confines the applicability of this approach to hardware platforms supporting block-sparse operations [40]. Chitty-Venkata *et al.* [117] similarly apply layer-wise MPS using a differentiable algorithm where suboptimal configurations are eliminated from the search space.

The approach proposed in this chapter differentiates from these techniques by integrating gradient-based optimization with *per-channel MPS* and pruning with *accurate hardware models*. Moreover, our method imposes no constraints on candidate bit-widths, unlike Bayesian Bits [114] and FracBits [118].

4.2 Proposed Method

Mixed-precision quantization and pruning serve as independent optimization strategies for reducing the complexity of a DNN. Typically, these techniques are implemented sequentially, one following the other. This stepwise methodology is not optimal for two main reasons: first, it necessitates a longer training time to achieve a fully optimized DNN, and second, it restricts the exploration of the search space since the optimization possibilities of the second technique are limited by the irreversible decisions made during the application of the first.

We introduced an innovative gradient-based approach that simultaneously performs mixed-precision quantization and pruning to address these limitations. This joint method enables a more reliable assignment of precision levels by also considering the potential pruning of specific sections of the DNN rather than only decreasing their precision when they do not worsen the task performance.

The rest of this section is organized as follows. The constructed search space is presented in Sec. 4.2.1. Then, a comprehensive description of the employed optimization technique is provided in Sec. 4.2.2. Sec. 4.2.3 discusses the developed complexity regularizers that enable steering the optimization towards solutions with lower costs. Sec. 4.2.4 outlines the adopted training methodology. Finally, Sec. 4.2.5 discusses various implementation aspects that ensure the obtained optimized models are compatible with the hardware and software used in mixed-precision inference.

4.2.1 Search Space

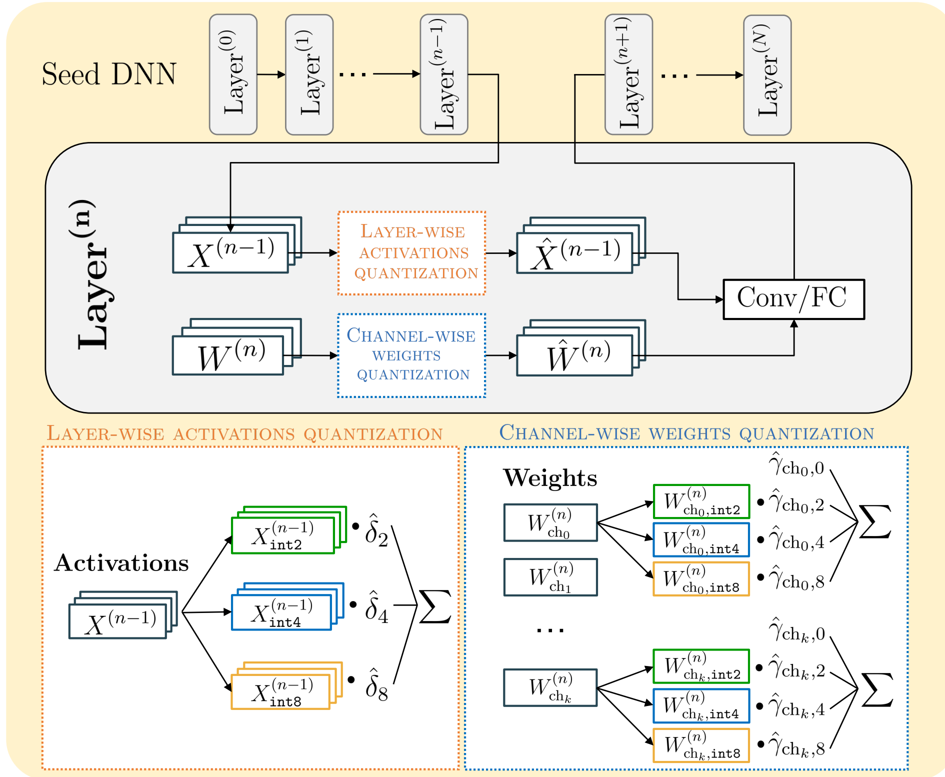

Fig. 4.1 Overview of the proposed joint quantization and pruning scheme for a DNN's generic layer.

An overview of the proposed method is depicted in Fig. 4.1. We denote P_W and P_X as the sets of explorable precisions for weights and activations. The search space encompasses all architectures derived from a reference DNN by quantizing its parameters and intermediate activations to every bit-width defined in P_W and P_X, respectively. In particular, *each weight channel* of a given layer can be independently

assigned to one of the available precision of P_W. On top of this scheme, pruning is integrated by introducing an extra candidate precision, $p_0 \in P_W$, which is to be intended as “0-bit quantization". Assigning 0 bits to all weights within a specific channel effectively sets them to zero, eliminating the channel’s information contributing to the DNN’s output. Consequently, the output activations of that channel become constant and equal to zero, making this approach functionally equivalent to pruning the channel.

To define the search space, each tensor in the DNN is coupled to a set of trainable bit-width selection parameters. Specifically, for the n-th layer with $C^{(n)}$out output channels and its corresponding weights $W^{(n)}$, we define a matrix of bit-width selection parameters $\gamma^{(n)} \in \mathbb{R}^{C^{(n)}\text{out} \times |P_W|}$, where each output channel is associated with a vector whose length matches the size of the weight precision set, as illustrated in the bottom right part of Fig. 4.1.

A similar procedure is applied to all intermediate activations within the DNN, but in this case, it is done with layer-wise granularity instead of channel-wise. The rationale for enabling channel-wise quantization for weights but maintaining layer-wise quantization for activations is detailed in Sec. 4.2.5. For each layer n, a parameter vector $\delta^{(n)} \in \mathbb{R}^{|P_X|}$ is assigned to its output activations, as depicted in the bottom-left box of Fig. 4.1. All bit-width selection parameters are optimized during the training phase, as explained in Sec. 4.2.2.

Crucially, the proposed method ensures that the same weight channels are pruned across pairs of layers in specific configurations. In particular, we consider two main classes of such configurations. First, when the outputs of two converging layers of a residual block are summed together, we must ensure that pruned channels are the same in both streams. A second class comprises depthwise-separable convolutions where a pointwise convolution follows a depthwise one. This consistency is achieved by sharing the precision selection parameters between these layers, ensuring that all pruned channels can be effectively removed from the network after optimization. For example, pruning a particular channel in one branch of a residual convolutional layer would result in minimal complexity reduction if the corresponding channel in the other branch remains untouched, as their addition would require retaining the channel in subsequent computations. In contrast, pruning the channel from both branches allows to reduce the number of feature maps to be processed by the following layers. Similarly, each kernel in a pointwise convolution generates a single output

feature map, which is then processed by a corresponding single depthwise kernel. Therefore, if an output channel from the pointwise operation is pruned, the associated depthwise channel can also be removed since it would process a “constant" feature map. In this scenario, only the pruned channels need to coincide, while channels with precisions different from 0-bit can be independently assigned to the depthwise and pointwise layers. However, our method shares the entire tensor of bit-width selection parameters between the two layers to avoid complicating the approach with multiple sets of masks. Although this slightly restricts the search space, empirical results demonstrate that it does not hinder the method’s effectiveness.

4.2.2 Optimization Method

We use gradient descent during training to jointly optimize the network’s weights and bit-width selection parameters. The training loss function comprises two components: one for improving the network’s task performance and another for reducing its complexity. This approach promotes selecting lower precisions (down to 0-bit) in less critical network parts. The objective function is thus the same as Eq. 2.11 and is reported here for the sake of clearness:

$$\min_{W,\theta} [\mathcal{L}_{\text{task}}(W,\theta) + \lambda \mathcal{R}(\theta)] \tag{4.1}$$

where W represents the network’s weights, and $\theta := \delta, \gamma$ denotes the bit-width selection parameters. The task loss term $\mathcal{L}_{\text{task}}(W,\theta)$ depends on both W and θ, while the regularization loss term $\mathcal{R}(\theta)$ only depends on the bit-width selection parameters and is designed to model the desired cost metric, such as size or latency, in a differentiable manner. The scalar hyper-parameter λ controls the relative weight of these terms, with higher λ values favoring more lightweight networks.

During the search phase, each weight channel or activation tensor in supported layers (e.g., convolutional and linear) is used to determine an effective tensor, as shown in Fig. 4.1. Given an activation tensor $X^{(n)}$ with associated bit-width selection parameters $\delta^{(n)}$, a discrete probability distribution is first derived. This yields a vector $\hat{\delta}^{(n)}$, where all elements are constrained within the $[0,1]$ range and sum to 1. We evaluate three sampling methods: softmax (SM), argmax (AM), and hard Gumbel-Softmax (HGSM). The sampling operation for bit-width selection

parameters, represented as $h(\delta)$, is defined for $i = 0, ..., |P_X|$ as:

$$\hat{\delta}_i = h(\delta)_i = \begin{cases} \frac{\exp(\delta_i/\tau)}{\sum_j \exp(\delta_j/\tau)}, & \text{SM} \\ \frac{\exp(\delta_i/\tau)}{\sum_j \exp(\delta_j/\tau)}, \tau \to 0, & \text{AM} \\ \frac{\exp[(\delta_i + \varepsilon)/\tau]}{\sum_j \exp[(\delta_j + \varepsilon_i)/\tau]}, \tau \to 0, & \text{HGSM} \end{cases} \tag{4.2}$$

where τ is the temperature which controls the smoothness of the distribution and $\varepsilon_i \sim \text{Gumbel}(0, 1)$ is an independent identically distributed sample drawn from the Gumbel distribution. Then, the effective tensor $\hat{X}^{(n)}$ is computed as:

$$\hat{X}^{(n)} = \sum_{p_x \in P_X} \hat{\delta}^{(n)}_{p_x} \cdot X^{(n)}_{p_x} \tag{4.3}$$

where X_{p_x} denotes the activation tensor represented with p_x bits. Consequently, the effective activation tensor is built as the linear combination of its floating-point versions quantized at all the considered precisions $p_x \in P_X$.

Similarly, it is done for network weights. Each k-th row of matrix $\gamma^{(n)}$ contains a vector of bit-width selection parameters for the k-th channel of the n-th layer. By applying one of the aforementioned sampling functions to each row $\gamma^{(n)}_k$, we derive $\hat{\gamma}^{(n)}_k = h\left(\gamma^{(n)}_k\right)$. The effective weight tensors are then calculated as:

$$\hat{W}^{(n)} = \sum_{p_w \in P_W} \hat{\gamma}^{(n)}_{p_w} \cdot W^{(n)}_{p_w} \tag{4.4}$$

Here, W_{p_w} refers to the weight matrix quantized at p_w bits, and $\hat{\gamma}^{(n)}_{p_w}$ includes C_{out} elements representing bit-width selection parameters for all output channels.

Using the effective activation and weight tensors, the output $Y^{(n)}$ of the n-th layer is determined as:

$$Y^{(n)} = l(\hat{X}^{(n-1)}, \hat{W}^{(n)}) \tag{4.5}$$

where l denotes the layer operation.

After training, each weight channel and activation is assigned a specific precision bit-width. The tensors are then replaced by their quantized versions at the precision corresponding to the highest bit-width selection parameter value. For the n-th layer's

activation:

$$X^{(n)} \leftarrow X^{(n)}_{p_x} \mid p_x := \underset{p_x \in P_X}{\operatorname{argmax}}(\hat{\delta}^{(n)}) \tag{4.6}$$

Conversely, for weights, each k-th channel is quantized at potentially different precisions:

$$W^{(n)}_k \leftarrow W^{(n)}_{k,p_w} \mid p_w := \underset{p_w \in P_W}{\operatorname{argmax}}(\hat{\gamma}^{(n)}_k) \tag{4.7}$$

Before implementing the search process described above, batch normalization parameters are folded into the preceding convolutional or linear layers. This ensures compatibility with hardware only with integer arithmetic units, as such units do not support floating-point batch normalization. Simulating full-integer inference during optimization ensures the deployed networks align more closely with their final implementation.

4.2.3 Complexity Regularizers

The methodology detailed in Sec. 4.2.1 and Sec. 4.2.2 is independent of the chosen cost metric used as a measure for the DNN's complexity ($\mathcal{R}$ in Eq. 4.1). Therefore, any differentiable measure of DNN cost can guide the training process, provided it can be incorporated into a gradient-based optimization loop. Commonly used cost functions include hardware-agnostic metrics for general DNN characteristics, such as model size or the number of MAC operations [119]. While model size correlates well with memory usage, the same is not valid for the number of MACs that often correlate poorly with actual performance metrics (e.g., inference latency or energy consumption) on real hardware [120]. Consequently, more precise regularizers can be devised that account for platform-specific support for combinations of weights and activation precisions and their corresponding efficiency.

The work presented in this chapter primarily focuses on model size as a hardware-agnostic cost metric. Additionally, we propose two hardware-specific regularizers to estimate inference latency on two deployment platforms: MPIC [3] and NE16 [84].

Size Regularizer

When the optimization goal is to minimize memory usage for storing model parameters, the activations' bit-widths have no impact since they only affect runtime

memory usage. Accordingly, the corresponding regularizer considers the effective weights' bit-width as the dot product of precision bit-widths and the associated selection parameters vector for each channel i in the n-th layer. For a convolutional layer, the regularizer can be expressed as:

$$\mathcal{R}^{(n)}(\hat{\gamma}^{(n)}) = C_{\text{in,eff}}^{(n)} K_x^{(n)} K_y^{(n)} \sum_{i=1}^{C_{\text{out}}^{(n)}} \sum_{p_w \in P_W} \hat{\gamma}_{i,p_w}^{(n)} p_w \tag{4.8}$$

where $K_x^{(n)}$ and $K_y^{(n)}$ represent the horizontal and vertical kernel sizes. The term $C_{\text{in, eff}}^{(n)}$ denotes the effective number of input channels, i.e., those not removed by pruning. E.g., in the case of sequentially connected layers, $C_{\text{in,eff}}^{(n)}$ is computed as $C_{\text{out}}^{(n-1)} - \sum_{i=1}^{C_{\text{out}}^{(n-1)}} \hat{\gamma}_{i,p_0}$. Using $C_{\text{in, eff}}$ instead of C_{in} in the cost function captures the benefit of pruning output feature maps, as this reduces the input size for subsequent layers, thereby reducing the number of weights required.

MPIC Regularizer

The latency cost of executing a DNN on MPIC (already introduced in Sec. 2.3.3) is quantified by using the Look-Up Table (LUT) from [3], which contains latency measured on fixed layer topologies. In particular, this LUT stores for all combinations of activations and weights bit-widths the number of MAC operations per cycle. Given a convolutional layer n, the MPIC regularizer $\mathcal{R}^{(n)}(\hat{\delta}^{(n-1)}, \hat{\gamma}^{(n)})$ is defined as:

$$\sum_{p_x \in P_X} \sum_{p_w \in P_W, p_w \neq 0} \frac{\mathcal{M}^{(n)}(\hat{\delta}_{p_x}^{(n-1)}, \hat{\gamma}_{p_w}^{(n)})}{\mathcal{T}(p_x, p_w)} \tag{4.9}$$

where $\mathcal{M}^{(n)}$ denotes the number of MACs executed in the n-th layer at the different p_x/p_w combinations and $\mathcal{T}$ represents the LUT. The number of MACs $\mathcal{M}^{(n)}(\hat{\delta}_{p_x}^{(n-1)}, \hat{\gamma}_{p_w}^{(n)})$ is computed as:

$$K_x^{(n)} K_y^{(n)} W^{(n)} H^{(n)} C_{\text{in,eff}}^{(n)} \hat{\delta}_{p_x}^{(n-1)} \sum_{i=1}^{C_{\text{out}}} \hat{\gamma}_{i,p_w}^{(n)} \tag{4.10}$$

Where the expression $\sum_{i=1}^{C_{\text{out}}} \hat{\gamma}_{i,p_w}^{(n)}$ computes how many output channels are theoretically processed at the precision level p_w, with $\hat{\gamma}_{i,p_w}^{(n)}$ indicating the quantization

parameter for channel i at weight bit-width p_w. The term $C^{(n)}_{\text{in, eff}}\ \hat{\delta}^{(n-1)}_{p_x}$ indicates what portion of input activations are theoretically processed at bit-width p_x. Finally, $W^{(n)}$ and $H^{(n)}$ denote the output dimensions of layer n - specifically its width and height. Essentially, Eq. 4.9 calculates the number of MAC operations executed for each combination of p_x/p_w precisions within the layer and uses the MACs-per-cycle values from the LUT to estimate the cost in terms of number of cycles. For the results presented in Table 4.2, we determine the latency using Eq. 4.9 and compute the energy consumption for a single inference based on the power measurements provided in [3], which were obtained with the core operating at a frequency of 250 MHz.

NE16 Regularizer

The cost model used to estimate the number of execution cycles on NE16 (ref. Sec. 2.3.3) for given precision assignments considers three primary factors: (i) The latency for loading weights, which is determined by the data STREAMER—a custom memory access engine capable of providing up to 288 bits of bandwidth; (ii) The time taken by the PE matrix to complete MAC operations, which depends on the number of output and input channels, and spatial dimensions, given that each PE can perform up to $3 \times 3 \times 32$ parallel MAC operations; (iii) The latency for storing results in the L1 memory, which is constrained by a bandwidth of 64 bits per cycle. The complete analytical model is available as open-source[1].

The total latency of each considered DNN has been computed, assuming the accelerator operates at a peak frequency of 370 MHz. Conversely, energy estimates for single inferences are not included, as no publicly available power measurements exist for this DNN accelerator.

Moreover, on this platform, we included a post-search refinement step for precision assignment to leverage the accelerator's parallelism fully. The precision assignment derived from gradient-based search may sometimes be misaligned with the hardware's parallel capabilities (e.g., assigning 33 channels to a given precision requires a second NE16 invocation for just one additional channel). In such cases, we employ a deterministic optimization step to selectively *increase* (but never decrease) the bit-width of some channels, reducing inference latency. This post-processing

[1] https://github.com/pulp-platform/dory/blob/master/dory/Hardware_targets/PULP/GAP9_NE16/Tiler/Ne16PerfModel.py

step does not require additional training and is completed in under 1 second on the considered models. Notably, it can be applied not only to NE16 but also to any hardware platform that parallelizes processing over output channels.

4.2.4 Training Procedure

The proposed method is executed in three phases: warmup, search, and fine-tuning. These three phases are similar to the ones discussed for PIT in previous Ch. 3 but are detailed here for this new approach. In the warmup phase, only the weights of the DNN are trained without including the bit-width selection parameters. The weights are optimized in floating point based solely on the task loss, with no regularization loss applied. The model resulting from this preliminary training is the initial point for the subsequent optimization stages.

The search phase is the core of the optimization process, where the bit-width selection parameters are trained alongside the DNN weights. The complexity-dependent regularization term ($\mathcal{R}$) is incorporated into the loss function during this phase. If softmax sampling is employed, the temperature parameter (τ) is progressively annealed to make the bit-width selection mimic an argmax operation. This gradual adjustment is beneficial when the distribution of sampled bit-width parameters is spread out but heavily influenced by the 0-bit precision. Without this step, inconsistencies could arise after discretization when channels are removed, as the layers might have adapted to non-zero contributions during training, which will not be present during inference.

After the search phase, the bit-width selection parameters are discretized (see Sec. 4.2.2) to assign a single precision to each considered tensor. Then, during the fine-tuning phase, the final quantized version of the DNN is trained to convergence, considering only the task loss term $\mathcal{L}_{\text{task}}$ as optimization target.

Weights rescaling

As expressed in Eq. 4.4, the effective weight tensor represents a weighted average of the quantized versions of the original weight matrix. Unlike other quantization approaches that approximate floating-point weights at varying precisions, the 0-bit quantization consistently outputs a constant zero value. Suppose the associated bit-

width selection parameter has a non-zero value at the start of the search phase. In that case, this will reduce the magnitude of the effective weight tensor compared to its post-warmup value, resulting in a notable drop in accuracy and suboptimal precision assignments. To address this issue, we rescale the weights from the post-warmup model to ensure that the 0-bit quantization does not reduce the effective weight tensor value. Specifically, at the beginning of the search phase, weight channels are adjusted as follows:

$$W_i^{(n)} \leftarrow \frac{W_i^{(n)}}{\sum\limits_{p_w \in P_W, p_w \neq 0} \hat{\gamma}_{i,p_w}^{(n)}} \tag{4.11}$$

Bit-width selection parameters initialization

Precision selection parameters for weights and activations (γ and δ) are initialized to address potential performance degradation at the start of the search phase so that higher bit widths are favored in the first epochs. Indeed, a uniform initialization of γ values would cause instabilities with discrete sampling methods, as such methods might disproportionately select the 0-bit precision across a large portion of the network or even entire layers, thereby interrupting gradient backpropagation. To mitigate this issue, the weights' bit-width selection parameters are initialized as follows:

$$\hat{\gamma}_{i,p_w}^{(n)} \leftarrow \frac{p_w}{\max\limits_{p \in P_W} p}, \ \forall p_w \in P_W \tag{4.12}$$

This approach assigns progressively smaller values to lower bit widths, ensuring that the 0-bit (pruning) option has the smallest associated sampling coefficient. A similar initialization strategy is applied to activations' δ parameters.

4.2.5 Implementation details

Our proposed method, which assigns to the DNN channels of each layer varying bit-widths, can be easily integrated with hardware supporting mixed-precision inference and compatible software libraries. As illustrated in the leftmost part of Fig. 4.2, after the search phase, the DNN layer's weight tensor channels can be quantized at different precisions without a specific order. To enable efficient execution on an edge device, these channels can be reorganized into groups based on the bit-width

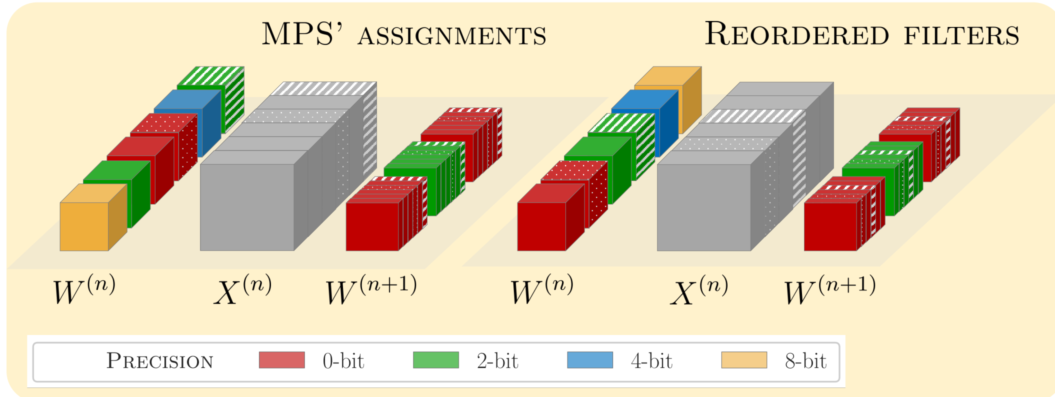

Fig. 4.2 Weights channels reordering by bit-width.

at which they are quantized, as shown in the rightmost part of the figure. This reorganization is performed offline, only once, before inference. It is important to notice that reordering the weight channels also affects the resulting output activations, as highlighted by the matching patterns in Fig. 4.2. Consequently, the weights of subsequent layers must be reordered correspondingly to preserve the correct mapping between input channels and weights.

Following this reordering, the original convolutional (or linear) layer can be divided into $|P_W|$ smaller, parallel sub-layers, as depicted by the colored shadows in Fig. 4.2. Since activations are quantized layer-wise, the outputs of these subdivided convolutions share the same precision and can be easily concatenated before being passed to the next layer. If activations were instead quantized on a channel-wise basis, the resulting mixed-precision outputs would be more challenging to store and efficiently access in memory for subsequent layers.

Crucially, our proposed channel-wise MPS leverages weight sharing, meaning all quantized weights combined as described in Eq. 4.4 originate from a single real-valued weight tensor. Increasing the number of candidate precisions incurs only a minimal memory overhead, as it requires adding a single quantization parameter per channel rather than duplicating the weight tensor. This overhead remains small regardless of the network size. Furthermore, due to the limited number of additional parameters to optimize, the training time overhead is comparable to layer-wise MPS with the same candidate precisions.

4.3 Experimental Results

4.3.1 Experimental Setup

We developed our approach using Python and PyTorch, building upon the PLiNIO library [32]. As candidate precisions for DNN weights, we consider 0, 2, 4, and 8 bits, which are well-supported by all the considered accelerators. The bit-width configurations for activations vary across different experiments. For activations, we adopt the PACT [81] quantization scheme, while weights are quantized using a symmetric min-max strategy.

Our method is evaluated against several SotA techniques. Specifically, we extended the PIT mask-based DNAS [34] described in Ch. 3 to support 2D convolutions. More details about the method extension are provided in subsequent Ch. 5 in Sec. 5.2.2. Additionally, we compare against EdMIPS [83], a SotA gradient-based mixed precision quantization technique, as well as a channel-wise MPS approach (referred to as MixPrec in the subsequent sections) where the 0-bit precision is not considered. Finally, we examine the sequential application of PIT and MixPrec to evaluate our method's effectiveness against the conventional practice of applying pruning and quantization separately. As outlined in Sec. 4.2.3, we consider as targets for our mixed-precision networks MPIC and NE16 platforms.

Our experiments span three distinct benchmarks: CIFAR-10 [50], Google Speech Commands (GSC) v2 [45], and Tiny ImageNet [51]. For CIFAR-10, we partition the dataset into training, validation, and test sets, comprising 66%, 17%, and 17% of the samples, respectively. The reference model is a custom ResNet with nine convolutional layers, following [121]. For GSC, following [45], we adopt the standard classification setup with 12 target labels, which include 10 original dataset keywords and two additional classes: “Silence" and “Unknown word". The dataset is split into training (85%), validation (10%), and test (5%) subsets. The reference model is the Depthwise Separable Convolutional Neural Network (DS-CNN) from [121]. For Tiny ImageNet, we divide the original training set into 90% training and 10% validation subsets, using the official validation set as the test set. The baseline model is a ResNet-18. All accuracy results are reported on the test sets.

4.3.2 Sampling Methods Comparison

Fig. 4.3 shows the results obtained when applying our method to the three con-

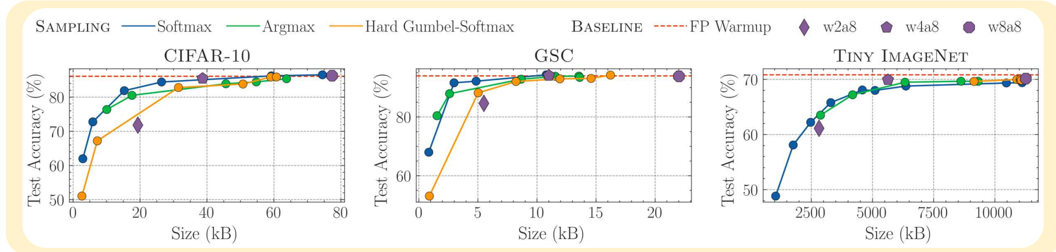

Fig. 4.3 Comparison of different sampling methods for our proposed approach on CIFAR-10, GSC, and Tiny ImageNet.

sidered benchmarks, using the size regularizer as the cost metric. Each curve point corresponds to a model trained with a distinct regularization strength (λ). Only the DNNs forming the Pareto front based on validation metrics are displayed. Each plot includes baseline models: the reference architecture trained in floating point and three fixed-precision versions with weights quantized at 2, 4, and 8 bits. Since activation quantization does not influence model size, activations are always quantized at 8 bits. Each color represents one of the different sampling methods presented in Sec. 4.2.2.

The leftmost plot illustrates CIFAR-10 results. Our approach yields three Pareto fronts, with model sizes ranging from a few kilobytes to 77.12 kB. Notably, the softmax sampling method consistently outperforms the others. At equivalent accuracy compared to the FP seed model, which requires 309.44 kB for parameters, we achieve an 80.86% model size reduction and a 23.43% reduction relative to the fixed-precision w8a8 baseline. Additionally, our method enables even greater model compression at the expense of minor accuracy loss. With only a 1.67% accuracy drop, we obtain a model of just 26.41 kB, representing a 91.46% reduction compared to the FP seed model. Moreover, at the same accuracy as the w2a8 reference model (71.81%), our model achieves a size of only 5.89 kB, thus reducing the seed and w2a8 models' sizes by 98.10% and 69.54%, respectively.

The middle plot reports results for GSC, where softmax sampling also delivers the best accuracy-model size trade-offs. With a 0.37% accuracy increase, we achieve an 87.76% size reduction relative to the FP seed model (88.06 kB). At the same accuracy level as the 8-bit baseline, we obtain a model that is 47.50% smaller.

Additionally, with a 2.35% accuracy drop compared to the FP seed, we achieve a model size of 2.98 kB, which is 96.62% smaller than the seed and 45.90% smaller than the w2a8 baseline, while still surpassing the latter in accuracy by 6.97%.

For Tiny ImageNet (rightmost plot), softmax sampling excels in the smallest model size region. In contrast, argmax and hard Gumbel-Softmax slightly outperform in accuracy for larger models, though the differences remain marginal. We do not achieve a model matching the FP seed accuracy on this complex benchmark, but we closely approach it. The highest-accuracy model reaches 70.08%, with a 75.48% size reduction from the FP seed (45.05 MB) and a 1.91% size reduction compared to the w8a8 baseline, while being slightly less accurate (-0.06%) than the latter. Although no solutions surpass the w4a8 reference model, our approach delivers notable improvements for higher regularization strengths. At an equivalent size to the w2a8 model, we improve accuracy by 2.39%. Moreover, with an additional 1.08% accuracy gain over w2a8, our model is 12.15% smaller.

These results show that Softmax is the most consistent sampling method, achieving optimal performance on CIFAR-10 and GSC while remaining competitive on Tiny ImageNet. Therefore, we employ Softmax sampling in all subsequent experiments.

4.3.3 State-of-the-art Comparison

Fig. 4.4 compares our method and SotA approaches. The architectures forming

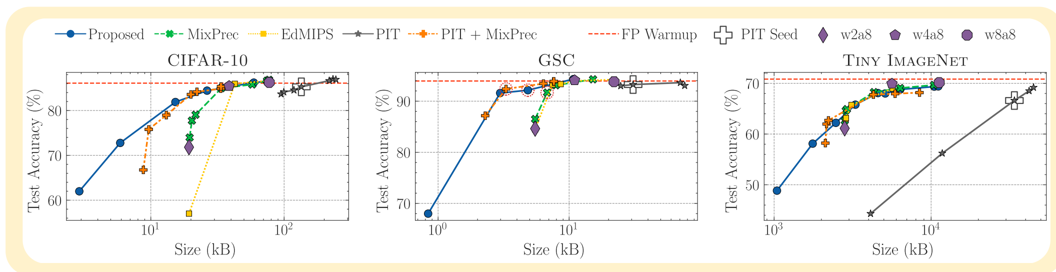

Fig. 4.4 Comparison of state-of-the-art approaches with the results obtained by our proposed method.

the Pareto curves are selected based on the trade-off between validation accuracy and model size. To evaluate the sequential application of PIT and MixPrec, we first generate a set of architectures by applying PIT to the floating point model, and then

we select one of these as a seed for MixPrec (indicated by a black plus symbol). Finally, we use this seed to conduct a new round of MixPrec searches by varying the λ parameter in Eq. 4.1 to obtain the final Pareto front.

In the leftmost plot of Fig. 4.4, we present results on CIFAR-10. In the size range between the w4a8 and w8a8 baselines, our method performs comparably to other MPS approaches. Differences emerge at smaller sizes, i.e., with higher regularization strengths. Notably, EdMIPS and MixPrec exhibit a hard size lower bound, which is associated with the assignment of the lowest bit-width to the entire DNN. This results in a size equivalent to the fixed-precision w2a8 baseline. However, accuracy varies depending on the weight optimizations applied during the search. Specifically, with EdMIPS, a significant accuracy drop occurs at iso-size with w2a8 due to training instability. Our method, leveraging pruning, overcomes this lower bound to discover Pareto-optimal solutions with even smaller sizes. Compared to the sequential PIT and MixPrec approach, our optimization strategy achieves equal or superior trade-offs by enabling more flexible precision assignments. In contrast, applying PIT first constrains MixPrec's search space, potentially leading to suboptimal results.

The middle plot of Fig. 4.4 illustrates the results on GSC. Here, EdMIPS and MixPrec yield nearly identical Pareto fronts. However, our method attains an accuracy of 91.60% with a model size of 2.98 kB. Indeed, at iso-accuracy, we achieve a 56.17% size reduction compared to the 6.79 kB model identified by MixPrec. Furthermore, our method performs comparably to the sequential PIT and MixPrec approach. This is because the reference FP model is significantly over-parametrized, meaning that PIT's pruning does not eliminate crucial channels that cannot be recovered later. Additionally, low-bit-width quantization preserves accuracy, allowing MixPrec to reduce precision with minimal performance degradation.

However, as highlighted in Table 4.1, our approach drastically reduces the total search time compared to the sequential PIT and MixPrec usage. Specifically, a single PIT epoch takes $1.8\times$ longer than a standard training epoch, while MixPrec and our approach incur a $4.3\times$ overhead. In contrast, obtaining a MixPrec seed using the sequential method requires first deriving PIT's full Pareto curve. Consequently, the overhead for a single solution is $(1.8N + 4.3)$ times the FP model's training time, where N represents the number of PIT-trained models, often exceeding the number of final Pareto points. For instance, in GSC, we trained four PIT models to construct the

Pareto front shown in Fig. 4.5, leading to a total overhead of 11.5×, approximately 2.7× higher than our joint optimization method.

Finally, the rightmost plot of Fig. 4.4 displays results on Tiny ImageNet. Similar to the other benchmarks, different methods yield comparable results for larger models, with the advantages of our approach becoming more evident for smaller architectures. With a model 12.15% smaller than the w2a8 baseline, our method achieves higher accuracy (62.21% vs. 61.13%). The sequential PIT and MixPrec method achieves a slightly greater size reduction (10.29%) at iso-accuracy compared to our solution. However, at 58.11% accuracy, our approach outperforms PIT + MixPrec by achieving a 16.96% smaller size. Moreover, considering the five PIT models forming the Pareto front, our method incurs an approximately 3.1× lower search time overhead.

4.3.4 Deployment

Fig. 4.5 presents the results on CIFAR-10 using our method in conjunction with

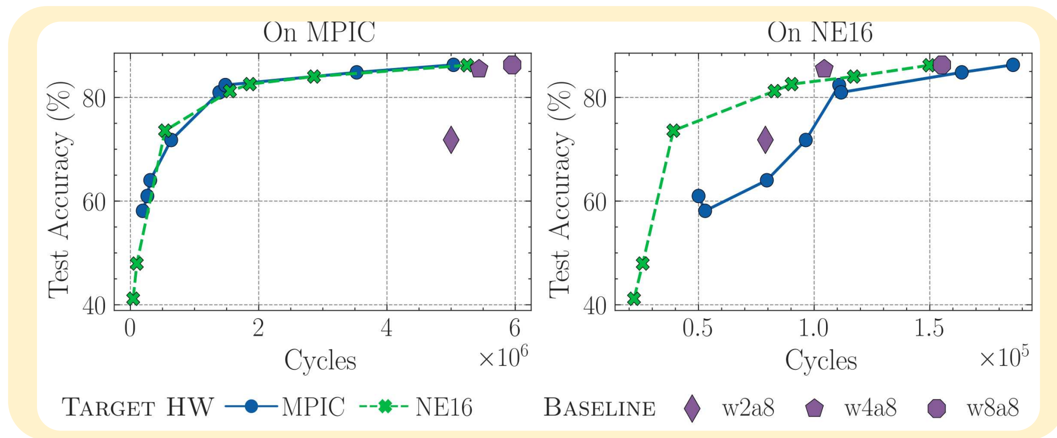

Fig. 4.5 Comparison of model performance in the accuracy-versus-cycles space for various cost regularization strategies on CIFAR-10. The plot on the left shows results when targeting MPIC for deployment, while the plot on the right corresponds to models optimized for deployment on NE16.

the latency cost models for MPIC and NE16, as described in Sec. 4.2.3. In this experiment, we kept activations at 8 bits to ensure a fair comparison between the models obtained with the two cost models, due to NE16 supporting only 8 bits for intermediate tensors. The green dashed curves depict the Pareto fronts obtained

using the NE16 latency cost model detailed in Sec. 4.2.3, whereas the blue curves correspond to results obtained with the MPIC cost model from Sec. 4.2.3.

Table 4.1 Average total training time speed-up of our approach with respect to the sequential application of PIT and MixPrec

Dataset	CIFAR-10	GSC	Tiny ImageNet
Average speed-up	3.9×	2.7×	3.1×

Table 4.2 Performance of baseline models quantized at fixed-precision and models trained with NE16 or MPIC regularizer on CIFAR-10.

			MPIC			**NE16**	
Model	**Size (kB)**	**Test Accuracy (%)**	**Cycles ($\times 10^6$)**	**Energy (μJ)**	**Latency (ms)**	**Cycles ($\times 10^3$)**	**Latency (ms)**
$\text{High}_{\text{MPIC}}$	74.98	86.28	5.038	108.46	20.15	186.014	0.50
$\text{Medium}_{\text{MPIC}}$	59.18	84.83	3.528	75.96	14.11	163.829	0.44
Low_{MPIC}	6.84	71.78	0.636	13.69	2.54	96.394	0.26
$\text{High}_{\text{NE16}}$	75.92	86.20	5.251	113.05	21.00	149.796	0.40
$\text{Medium}_{\text{NE16}}$	47.12	84.02	2.862	61.62	11.45	117.144	0.32
Low_{NE16}	12.90	73.59	0.540	12.01	2.16	39.119	0.11
w8a8	77.36	86.26	5.953	128.17	23.81	155.241	0.42
w4a8	38.68	85.46	5.435	117.03	21.74	104.361	0.28
w2a8	19.34	71.81	5.001	107.66	20.00	78.921	0.21

The leftmost plot of Fig. 4.5 illustrates the trade-off between accuracy and execution cycles when deploying models on MPIC hardware. Conversely, the rightmost plot shows results for models deployed on the NE16 accelerator. To emphasize the importance of hardware awareness, we also include results for models optimized using the “incorrect" cost model for each target. Namely, we use the cost model of MPIC for NE16 and vice-versa. The impact of using a mismatched cost model is minimal on MPIC, given its flexibility as a CPU-based platform. Conversely, the benefits of aligning the complexity metric with the hardware are substantial for NE16. For instance, at iso-accuracy with the w2a8 baseline, employing the MPIC cost model results in a 22.14% increase in cycles, whereas using the NE16 cost model achieves a 50.43% reduction. This disparity arises from NE16’s more complex dataflow and spatial parallelism, which favor assigning the same bit-width to entire channel chunks. Deviations from this strategy can waste computational bandwidth, reducing latency gains. The NE16 cost model accounts for these factors, while the MPIC model does not, leading to suboptimal solutions when applied to NE16.

A detailed breakdown of models sampled from the Pareto curves in Fig. 4.5 is provided in Table 4.2. Specifically, for each target hardware, we select the Pareto-optimal model with the highest cycle count (High), the fastest model achieving over

70% validation accuracy (Low), and an intermediate model (Medium) with cycle count closest to the mean of High and Low. We report accuracy, model size, execution cycles, latency, and energy per inference, including baseline fixed-precision models for reference.

The $\text{High}_{\text{MPIC}}$ model achieves comparable latency and energy costs to the reference w2a8 on MPIC while improving test accuracy by 14.47% through pruning. However, deploying this model on NE16 results in a 136% latency increase due to the cost model mismatch. In contrast, the Low_{NE16} model improves test accuracy by 1.78% while reducing latency by 89.20% on MPIC and 50.43% on NE16 relative to the w2a8 baseline.

The impact of cost models is particularly pronounced in *tiny* DNNs, where mismatched precision assignments incur significant relative overheads. For this reason, while our method scales to larger networks and datasets, its hardware awareness is especially beneficial for small-scale models.

4.3.5 Ablation studies

Models analysis

Fig. 4.6 presents the distribution of assigned precisions across the weight channels

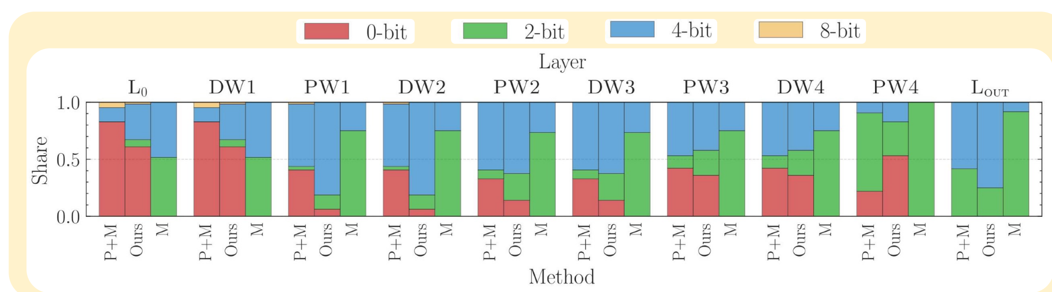

Fig. 4.6 Bit-width distributions of weight channels across three models trained on GSC using different strategies: PIT combined with MixPrec (P+M), our joint MixPrec and pruning approach (Ours), and MixPrec alone (M), all with size regularization applied. The input and output layers are labeled as L_0 and L_{out}, respectively, while DW and PW refer to depthwise and pointwise convolutional layers.

of each layer. We analyze three models trained on the GSC benchmark using the size cost model, comparing our approach with MixPrec and MixPrec + PIT. We selected

the models from the Pareto fronts in Fig. 4.4, indicated by red circles. As expected, combining PIT with MixPrec results in more pruned channels than our method across nearly all layers. This occurs because PIT reduces the DNN's complexity exclusively by removing entire channels, leading to a significantly smaller search space. To compensate, the remaining channels are quantized at higher bit widths by MixPrec to preserve model capacity. In contrast, our method relies less on pruning and instead leverages lower bit widths to reduce complexity. This is made possible by a larger search space, unrestricted by prior optimization steps from another method. The standalone application of MixPrec —without any pruning mechanism— produces models with more parameters quantized at 2 bits. This is expected, as 2-bit precision represents the lowest complexity option available to MixPrec at high regularization strengths. However, many of these channels could be entirely removed without impacting accuracy, as demonstrated by our approach.

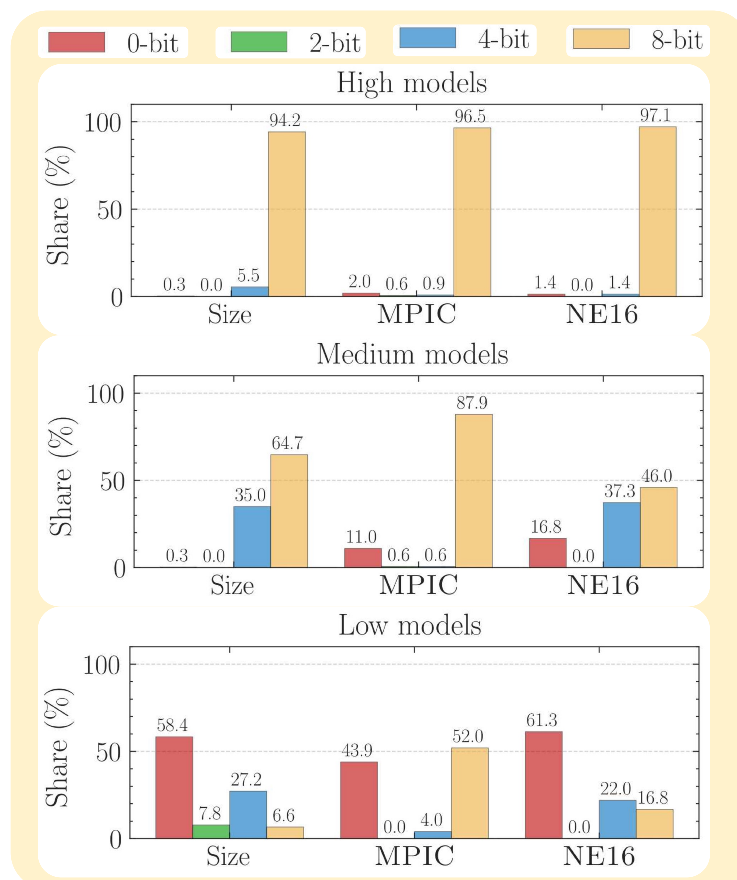

Fig. 4.7 Weights bit-widths distributions on CIFAR-10 for models with different regularizers employed in the training objective and with different complexity.

Fig. 4.7 illustrates how different cost regularizers influence weight precision assignments. We examine three cost models, i.e., Size, MPIC, and NE16, while keeping activations fixed at 8 bits for a fair comparison. We select three DNNs from the Pareto fronts on CIFAR-10 for each cost model, following the same rationale as in Table 4.2. In all three “High” models (top plot), the majority of weight parameters are quantized at 8 bits, the highest available precision, as expected. In “Medium” models (middle plot), a few to none parameters are assigned 2-bit precision. This is because 2-bit quantization offers limited representational capacity while incurring a non-negligible computational cost. Instead, 4-bit channels constitute a significant portion of the model, particularly for the Size and NE16 regularizers. This is due to their lower cost relative to 2-bit weights while providing substantial accuracy benefits. For “Low” models (bottom plot), 4-bit precision is more frequently assigned than 8-bit, except when using the MPIC cost model. A more detailed analysis reveals that 8-bit precision is typically favored only in the last layer, a trend commonly observed in prior works [83, 118]. The Size regularizer is the only one that assigns some channels to 2-bit precision. Meanwhile, the MPIC cost model prioritizes pruning while keeping most remaining weights at 8 bits, as the cost difference between 8-bit and lower precisions is insufficient to justify further reductions. The NE16 cost model, in contrast, promotes a more balanced distribution between 4-bit and 8-bit precision but completely avoids 2-bit assignments. This is because NE16’s cost scaling is not linear with the number of output channels. Each processing element (PE) processes groups of 32 output channels (Sec. 4.2.3). Consequently, running a single channel at a given precision incurs the same cost as running 32 channels, meaning that for optimal latency, NE16 should process at least 32 2-bit filters simultaneously. However, this would lead to accuracy degradation, making it an undesirable trade-off that the optimization naturally avoids.

Impact of activations’ quantization

All previous results were obtained by fixing the activations’ precision at 8-bit, either to enable a fair comparison between the MPIC and NE16 cost models or because reducing it was not beneficial, i.e., when optimizing for model size. Fig. 4.8 shows the results of applying our MPS method to activations, allowing layer-wise selection among 2, 4, and 8-bit precision. The resulting Pareto curve (orange) is compared against weights-only MPS with 8-bit activations (blue). Fixed-precision

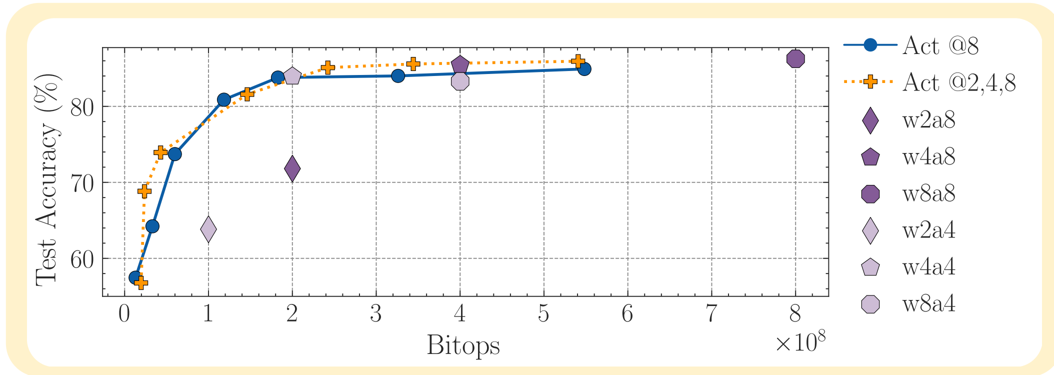

Fig. 4.8 Pareto fronts of architectures obtained on CIFAR-10 by either assigning the precision layer-wise from the set $P_X = \{2,4,8\}$ or quantizing the activations at 8 bits.

baselines are also included, except for the w*a2 models, which exhibit unacceptably low accuracy (<40%) for any practical computer vision task. The results are reported on CIFAR-10, using the bitops cost model [83], a hardware-agnostic latency proxy.

As expected, reducing activations' precision below 8-bit generally improves the trade-off. The most significant gain is observed at the second point from the left on both curves: in this case, reducing activations' precision improves accuracy by 4.61% while decreasing bitops by 28.51%. However, for other cases, the average accuracy difference between models with similar bitops is less than 1%. This contrasts with the baselines, where w4a4, for example, achieves a substantial trade-off by maintaining the same bitops as w2a8 while improving test accuracy by 12.14%. The observed differences are also smaller than those reported by prior MPS methods [83, 106]. We attribute this to the pruning capabilities of our method. When pruning is unavailable, reducing activations' precision is highly beneficial. Conversely, when pruning weight channels is an option, maintaining 8-bit precision while reducing the number of features provides a comparable representational capacity at a lower computational cost. Our method uncovers insights like this, which depend on the precision set, DNN model, and quantization scheme. Such findings could be instrumental in guiding future hardware design decisions.

Chapter 5

DNN Optimization with Multiple Hardware Constraints

The gradient-based optimization techniques presented in this thesis up to this point are all characterized by the solution of an optimization problem in the form of Eq. 2.11. In this equation, the overall loss function accounts either for task performance ($\mathcal{L}$) or for a linear combination of task performance and a cost metric ($\mathcal{L} + \lambda \mathcal{R}$), where $\mathcal{R}$ may express the total number of parameters of the model, the number of OPs per inference, etc. Under this formulation, identifying a model that satisfies a specific constraint (e.g., fixed parameter count) requires iteratively adjusting the weighting factor λ to find a good trade-off between the competing loss terms. As a result, despite being labeled as "one-shot" current gradient-based approaches are not capable of producing a neural network that simultaneously satisfies multiple hardware constraints in a single pass. Nonetheless, an optimal model should simultaneously achieve high accuracy, fit in the device's limited memory (down to a few MBs of Flash and RAM for MCU), and match the throughput or real-time latency imposed by the application.

While recent approaches have made progress toward true one-shot NAS [122–124], they typically focus on optimizing a single cost objective. These techniques have not been demonstrated in practical scenarios involving multiple, simultaneous constraints [74, 76].

This chapter proposes a new formulation of the optimization problem of Eq. 2.11 that permits finding a DNN that achieves high accuracy in a single training run

and respects both a maximum latency constraint and a total model size constraint. Then, we show how the approach is extended to more fine-grained layer-level constraints, that allows to tailor the DNN architecture to exploit the target hardware's memory hierarchy optimally. We show how this approach can be applied to both path-based [79] and mask-based [34] DNAS.

This chapter first provides an overview of the related works in Sec. 5.1, highlighting SotA techniques. Then, Sec. 5.2 details the proposed optimization method. Finally, Sec. 5.3 presents the experimental validation of the method.

Additionally, this chapter describes how to extend to 2D CNNs the channel-pruning scheme of PIT described in Ch. 3 for 1D TCNs. This pruning scheme will be then used in Sec. 7.2 of Ch. 7 to compress SotA CNNs taken from literature to be deployed on real hardware set-ups. In particular, Sec. 7.2 deals with a computer vision task where images collected with a nano-drone are processed with the CNN to estimate the pose of a moving person in the field of view. This information is then used to control the drone to maintain the person head in the center of its field of view.

The work described in this chapter has been published in [125, 126].

5.1 Related Works

Early methodologies [127–130] address integrating hardware constraints into the optimization process using EA. These algorithms evolve a population of architectures to generate an optimal Pareto front within the performance-complexity domain. Notably, CNAS [130] formulates the complexity objective as an aggregate of terms, wherein each term represents a different cost metric. These terms are calculated as the maximum value between zero and the cost exceeding a predefined target, scaled by a hyper-parameter that modulates the penalty magnitude. Among RL techniques, MNASNet [37] aims to enforce constraints by multiplying the task quality reward term Q_{task} (e.g., accuracy) by the ratio of the network's latency, empirically measured from a complete deployment to a corresponding target value. A hyper-parameter controls the exponent of this ratio, assuming distinct values based on constraint satisfaction. Consequently, the cost is a penalty term ($\lambda \neq 0$) when $\mathcal{R} > T$, and conversely, is zero ($\lambda = 0$) when $\mathcal{R} \leq T$. TuNAS [131], conversely, defines the cost term as the absolute deviation of the ratio between the cost and the target from

the ideal value of one. Latency is estimated utilizing a LUT-based methodology, and the cost component is incorporated into the reward function after scaling by a strength factor λ, constrained to non-positive values. Notably, this formulation imposes an equivalent penalty on models exceeding the constraint and those falling short of the target by the same cost. Finally, AutoTinyML [132] employs Bayesian Optimization (BO) to define the solution subspace that satisfies specified constraints and to optimize both the architecture and its parameters.

All the above approaches are iterative and thus suffer from the scalability issues extensively discussed in this manuscript. Also in this case, the gradient-based method represents a more lightweight solution. LightNAS [122] is a hybrid approach combining gradient-based optimization with EA. It defines the regularization term as the normalized difference between a model's latency and a target value, where normalization is achieved by rescaling with the target latency. Latency is estimated through an MLP model trained on empirical measurements from the target hardware. The hyper-parameter λ is adaptively adjusted during training via gradient ascent to ensure the constraint is met. Specifically, if the measured latency exceeds the target, λ is incremented to prioritize the cost loss term in subsequent optimization iterations, and conversely, if the measured latency is less than the target, λ is decremented. Similarly, UDC [124] employs a regularization formulation to constrain the number of parameters, but the λ value is constant. HardCoRe-NAS [123] integrates gradient-based optimization with Integer Linear Programming (ILP), employing the Block Coordinate Stochastic Frank-Wolfe Algorithm to direct the search towards an optimal architecture within a feasible region that satisfies latency constraints.

5.2 Proposed Method

As said, the gradient-based optimization approaches following the formulation of Eq. 2.11 exhibit several limitations. Indeed, the optimization space for real-world DNNs is multidimensional and involves intricate inter-dependencies, which a single cost metric modeled by $\mathcal{R}$ cannot adequately capture. Focusing solely on one metric, such as the number of operations, memory footprint, or a differentiable approximation of energy consumption or latency [31, 35], inherently ignores the others, leading to sub-optimal explorations.

Moreover, in Eq. 2.11, complexity is treated as an additional objective to minimize rather than a constraint. This approach conflicts with the primary goal of most designers, who aim to achieve the most accurate DNN that satisfies specific cost conditions. From the designer's point of view, all relevant cost metrics must be within specified bounds, allowing the DNN to fit within memory constraints, respect real-time requirements, and meet expected battery life, among other criteria. However, they are not concerned with reducing costs beyond these limits.

Achieving this with Eq. 2.11 is challenging, even considering a single cost metric. The formulation does not limit the value of $\mathcal{R}$, allowing the optimizer to continue reducing $\mathcal{R}$ to improve its loss function, potentially at the expense of accuracy, even after reaching the desired target. Consequently, designers must repeat the search process multiple times (as done in Ch. 3 and Ch. 4), adjusting the regularization strength λ until they find a model with the desired cost and high accuracy. Although recent studies have proposed alternative formulations to incorporate cost targets in a DNAS search [122, 124], they still focus on a single cost metric and attempt to match it precisely, which may not be optimal.

This chapter presents a novel and flexible DNAS formulation and training mechanism to address these issues, named **D**NAS **U**nder **C**ombined **C**onstraints **I**n **O**ne-Shot (**DUCCIO**). DUCCIO offers a flexible framework that can be easily integrated into any DNAS pipeline with only minimal, yet essential, adjustments. We demonstrate DUCCIO's applicability in two key use cases:

1. Identifying architectures that adhere to both a storage limit and a cap on the number of operations, i.e., an approximate measure of latency.
2. Refining architectures under per-layer size constraints to fit within the fastest available memory on the target hardware. Then balancing performance gains and minimal impact on accuracy.

We demonstrate DUCCIO's flexibility by applying it to two different state-of-the-art gradient-based optimization methods with distinct optimization mechanisms: the path-based DNAS DARTS [79] and mask-based DNAS PIT [34], previously discussed in Ch. 3. As in the rest of this thesis, we focus on CNNs, although DUCCIO can also be applied to other types of DNNs.

5.2.1 Path-based DNAS

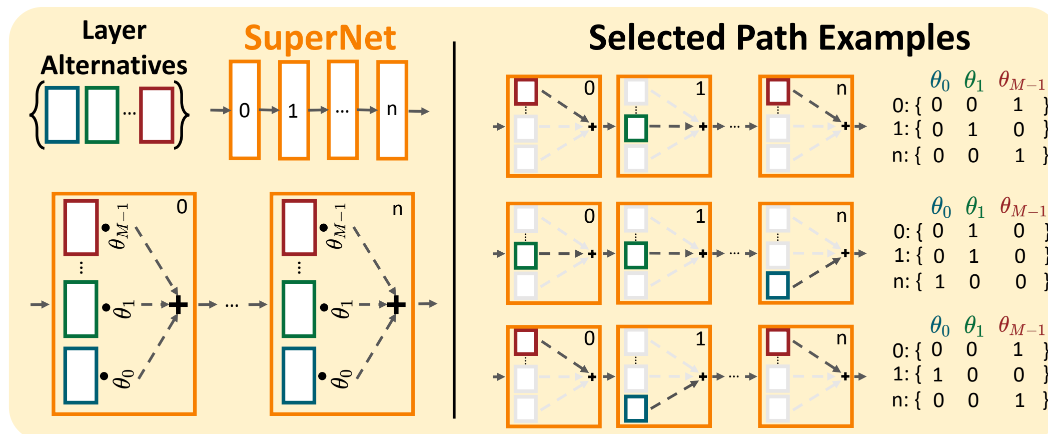

Fig. 5.1 Target path-based DNAS method

The first DNAS method we examine is inspired by DARTS [79], which employs a supernet and thus is fully classified as a path-based approach.

Fig. 5.1 illustrates its optimization scheme. The supernet is constructed from a standard CNN, where each convolutional layer $L^{(n)}$ is substituted with a module comprising a set of M different layer alternatives $\mathcal{M}^{(n)} = \{l_i^{(n)}\}_{m=0}^{M-1}$. Within each module, all layers share the same input, while each output tensor is associated with an additional trainable parameter $\theta_i^{(n)}$. These parameters $\theta_i^{(n)}$ are used to select among alternatives after the search. The supernet is integrated into a standard training loop, where the weights of each layer and the corresponding θ are trained together, using the objective function detailed in Sec. 5.2.3.

Our path-based DNAS differs from DARTS in two main ways. In line with recent NAS literature [133, 33], we utilize the Gumbel-Softmax [134] sampling strategy instead of the standard Softmax employed in [79]. Additionally, we adopt a discretized sampling strategy, where only one path is chosen in each training iteration. In practice, the output of the supernet module is constructed as follows:

$$y^{(n)}(x) = \sum_{i=0}^{M-1} g(\theta_i^{(n)}) \cdot l_i^{(n)}(x) \tag{5.1}$$

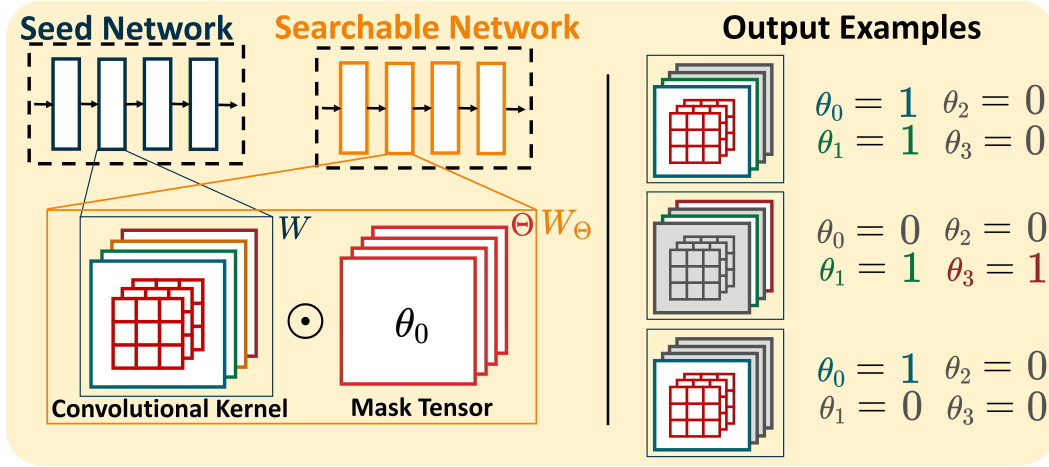

Fig. 5.2 Target mask-based DNAS method.

where $g(\theta_i^{(n)})$ are discretized Gumbel-Softmax coefficients:

$$g(\theta_i^{(n)}) = \text{onehot}_i\left(\frac{\exp(\theta_i^{(n)} + \varepsilon)}{\sum_i \exp(\theta_i^{(n)} + \varepsilon)}\right) \tag{5.2}$$

Here, $\varepsilon \sim \text{Gumbel}(0,1)$ denotes independent and identically distributed samples from the Gumbel distribution [134], and the one-hot function selects the maximum sampled alternative by setting it to 1 and all others to 0. To address the non-differentiability of the one-hot operation, we use an STE [95], which substitutes the one-hot function with an identity mapping during the backward pass. This technique enables gradient propagation through the discrete sampling process. As elaborated in Sec. 5.2.4, discretized sampling is crucial for respecting cost constraints.

Upon completion of training, the optimized architecture is derived by selecting, for each supernet module, the alternative corresponding to the largest $\theta_i^{(n)}$.

5.2.2 Mask-based DNAS

The second DNAS method we examine is a straightforward yet effective mask-based approach. Starting from a seed CNN, it performs a detailed optimization of the number of output channels for all convolutional layers. This DNAS, illustrated in Fig. 5.2, extends the channel-search method initially proposed in PIT [34] and described in Ch. 3 to 2D CNNs.

In this approach, a searchable model, which the DNAS can optimize, is constructed by pairing each convolutional weight tensor in the seed $W^{(n)}$ with a trainable mask vector $\theta^{(n)}$. The dimensions of $W^{(n)}$ are $C_{out}^{(n)} \times K_x^{(n)} \times K_y^{(n)} \times C_{in}^{(n)}$, where $C_{in}^{(n)}$ and $C_{out}^{(n)}$ represent the input and output channels, and $K_x^{(n)}$ and $K_y^{(n)}$ are the horizontal and vertical kernel sizes, respectively.

The corresponding $\theta^{(n)}$ contains $C_{out}^{(n)}$ elements, one for each output channel, and is used to prune the least significant channels during the search. Consequently, a masked weight tensor $W_\Theta^{(n)}$ is constructed as follows:

$$W_\Theta^{(n)} = W^{(n)} \odot \mathcal{H}(\theta^{(n)}) \tag{5.3}$$

where $\odot$ is the Hadamard product, and $\mathcal{H}$ is a Heaviside step function centered in 0 (fixed threshold $t_h = 0$). This function is used to binarize $\theta^{(n)}$ (0 if $\theta_i^{(n)} < t_h$, 1 otherwise). Each binarized mask element is multiplied with a slice of weights $W_i^{(n)}$ along the $C_{out}^{(n)}$ dimension, which includes all weights of the i-th convolutional filter. Thus, when $\mathcal{H}(\theta_i^{(n)}) = 0$, the i-th output channel is effectively removed from the network. Conversely, the channel is retained if $\mathcal{H}(\theta_i^{(n)}) = 1$.

The constructed network is inserted into a conventional training loop, where both W and θ are optimized jointly using DUCCIO's objective. Binarization ensures that each training iteration samples a single discrete architecture (refer to the examples on the right in Fig. 5.2). To maintain gradient flow during backpropagation, an STE substitutes the non-differentiable Heaviside function with an identity mapping.

Finally, the optimized model is exported by permanently removing all channels corresponding to $\theta_i^{(n)} < t_h$, thereby reducing the size of each convolutional layer accordingly.

5.2.3 Multi-Constraint Loss Formulation

As outlined in Sec. 5.1, SotA DNAS tools [35, 31, 33] mainly optimize a variation of Eq. 2.11, which combines the task-specific loss $\mathcal{L}$ with a regularization term $\mathcal{R}$. The latter represents a single cost metric as a differentiable function of the architectural optimization parameters (θ). The interpretation of $\mathcal{R}$ differs across approaches. Some methods [31, 33] define it in terms of total parameters or OPs in specific layers (e.g., all Convolutional layers), while others [35] approximate end-to-end latency

or energy consumption of the obtained DNNs. However, these approaches have inherent limitations, as noted earlier in this section:

1. They optimize only a single cost metric, while real-world systems impose multiple constraints.
2. They do not ensure that the estimated cost $\mathcal{R}$ stays below a predefined threshold.
3. Even if the target is met, $\mathcal{R}$ remains part of the objective function, unnecessarily leading the optimizer to favor lower-cost models.

Consequently, achieving the desired cost-accuracy balance requires sweeping λ across multiple values.

Recent works such as UDC [124] and LightNAS [122] introduce an explicit cost target T in the regularization term. These methods guide the optimizer to minimize the gap between estimated cost and target by adding a term proportional to $|\mathcal{R}(\theta) - T|$ in the loss function.

While these strategies mark progress, they do not fully resolve the issues above. First, they still focus on a single cost dimension, leaving issue 1) unaddressed. Similarly, issue 3) persists, as once $\mathcal{R}(\theta)$ falls below T, the added loss term counteracts this effect, pushing the optimizer towards higher-cost models. Some works [131] argue that this is beneficial, as more complex models tend to be more accurate. However, our experiments contradict this assumption, demonstrating that allowing NAS to explore cost variations within constraints yields superior results. In some cases, lower-complexity models perform as well as those closely adhering to the target.

DUCCIO overcomes these limitations by proposing the following objective function:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \sum_{j=0}^{J} \lambda_j \max(0, \mathcal{R}_j(\theta) - T_j) \tag{5.4}$$

This formulation introduces two key enhancements. First, it supports an arbitrary number (J) of cost metrics ($\mathcal{R}_j$) and constraints (T_j), each with its regularization coefficient. Second, by incorporating a max function, the penalty applies only when a cost metric exceeds its constraint. Conversely, if $\mathcal{R}_j(\theta) < T_j$, the respective term vanishes from the loss function, ensuring that NAS parameter gradients remain

unaffected. During DUCCIO searches, the coefficients λ_j are dynamically adjusted to ensure all constraints $\mathcal{R}_j(\theta) < T_j$ hold at the end of training, as detailed in Sec. 5.2.4.

Notably, DUCCIO allows for the inclusion of cost objectives in its loss function (e.g., $+\lambda_j\mathcal{R}_j(\theta)$). This flexibility is useful when a cost metric, such as energy consumption, lacks a strict constraint but should be explored across the Pareto front for accuracy trade-offs. In this scenario, DUCCIO behaves like a conventional DNAS. However, this study focuses on one-shot DNAS, treating all cost dimensions as constraints.

The following sections present two practical applications of DUCCIO's loss formulation.

Multiple Global Constraints

A straightforward application of DUCCIO is the enforcement of multiple global constraints on the entire model under optimization. System designers frequently need to ensure that i) the total DNN size remains within the storage limitations of the target platform and ii) the total inference latency satisfies a real-time requirement. In such cases, Eq. 5.4 can be written as:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \lambda_0 \max(0, \mathcal{O}(\theta) - T_o) + \lambda_1 \max(0, \mathcal{S}(\theta) - T_s) \tag{5.5}$$

Where $\mathcal{S}$ represents the DNN's storage footprint as a function of the NAS parameters θ, and $\mathcal{O}$ serves as a proxy for latency. To estimate the model's total size, we sum the sizes of all NAS target layers:

$$\mathcal{S}(\theta) = \sum_{n=0}^{N-1} \mathcal{S}^{(n)}(\theta) \tag{5.6}$$

The formulation of $\mathcal{S}^{(n)}$ varies depending on the chosen DNAS strategy. In our path-based approach, it is calculated by taking the number of weights associated with each supernet option and scaling it by the respective NAS sampling coefficient:

$$\mathcal{S}^{(n)}(\theta) = \sum_{i=0}^{M-1} g(\theta_i^{(n)}) \cdot \mathcal{S}_i^{(n)} \tag{5.7}$$

For instance, in the case of a standard convolution, $\mathcal{S}_i^{(n)} = C_{in}^{(n)} \cdot C_{out}^{(n)} \cdot K_x^{(n)} \cdot K_y^{(n)}$, while for a sequence of DepthWise and PointWise convolutions, as found in MobileNets [30], the size is given by $\mathcal{S}_i^{(n)} = C_{in}^{(n)} \cdot K_x^{(n)} \cdot K_y^{(n)} + C_{in}^{(n)} \cdot C_{out}^{(n)}$.

For mask-based DNAS, model size is estimated as the number of weights in non-masked filters. Taking a standard convolution as an example:

$$\mathcal{S}^{(n)}(\theta) = C_{in}^{(n)}(\theta) \cdot C_{out}^{(n)}(\theta) \cdot K_x^{(n)} \cdot K_y^{(n)} \tag{5.8}$$

where C_{out} depends on θ since channels with $\theta_i^{(n)} < t_h$ are effectively pruned. Consequently, C_{in} is indirectly affected due to reductions in C_{out} in preceding layers. For a sequential model, this results in $C_{in}^{(n)} = C_{out}^{(n-1)}$.

Regarding latency, a simple approach is to count the MAC operations in each target layer. For convolutional layers, this can be formulated as:

$$\mathcal{O}(\theta) = \sum_{n=0}^{N-1} \mathcal{S}^{(n)}(\theta) \cdot O_x^{(n)} \cdot O_y^{(n)} \tag{5.9}$$

Where $O_x^{(n)}$ and $O_y^{(n)}$ denote the width and height of the output feature map in the n-th convolutional layer. Although $\mathcal{O}$ and $\mathcal{S}$ are correlated, optimizing one does not necessarily optimize the other. In most DNN architectures, output feature maps tend to shrink due to pooling and strided convolutions while the number of channels increases. As a result, initial layers typically contribute more to latency, whereas final layers impact model size more significantly.

In this work, we adopt this simplified latency model, as it demonstrates a strong correlation with actual latency on our target platform. However, DUCCIO remains independent of the specific cost formulations and is compatible with more accurate differentiable latency estimations, such as those proposed in [35, 122].

Layer-wise Constraints

Another common challenge DNN system engineers face is dealing with models that are not optimally sized for the available hardware. For instance, a DNN may contain one or more layers that slightly exceed a specific memory level (e.g., L2), leading to costly data transfers of inputs, outputs, or weights from a higher memory level (e.g., L3) [102]. With DUCCIO, we can perform a one-shot optimization that selectively

shrinks only these critical layers while maximizing accuracy by formulating the objective as follows:

$$\min_{W,\theta} \mathcal{L}(W;\theta) + \sum_{n \in Crit.} \lambda_n \max(0, \mathcal{M}^{(n)}(\theta) - T_m) \tag{5.10}$$

Where $Crit.$ represents the set of critical layers, i.e., those slightly exceeding a given memory level, and T_m denotes the available space at that level (i.e., its total size minus a constant offset accounting for code, other variables, etc.). The total memory consumption of the n-th layer, $\mathcal{M}^{(n)}(\theta)$, for a convolutional operation is given by:

$$\mathcal{M}^{(n)}(\theta) = \mathcal{S}^{(n)}(\theta) + I_x^{(n)} I_y^{(n)} C_{in}^{(n)}(\theta) + O_x^{(n)} O_y^{(n)} C_{out}^{(n)}(\theta) \tag{5.11}$$

Here, the three terms account for the memory required by parameters, inputs, and outputs, respectively, where $I_x^{(n)}$ and $I_y^{(n)}$ are the input feature dimensions. The rightmost two terms depend on θ through C_{in} and C_{out} in the mask-based DNAS method, whereas they remain fixed in the path-based approach.

Unlike Eq. 5.5, which imposes constraints on aggregate metrics across the entire DNN, Eq. 5.10 applies constraints at the individual layer level. Furthermore, while the equation assumes a common target T_m for all critical layers for simplicity, it can be extended to support different targets, ensuring that some layers fit within L1 memory, others within L2, etc.

To our knowledge, existing SotA DNAS methods have not considered such fine-grained, hardware-aware constraints. As shown in Sec. 5.3, adapting each layer to the memory hierarchy can lead to significant speedups with only minor accuracy trade-offs.

5.2.4 Training Procedure

Algorithm 2 outlines the DUCCIO training procedure, which, like other gradient-based optimization approaches, consists of three distinct phases: warmup, search, and fine-tuning. These three phases are similar to those presented in Ch. 3 and Ch. 4 but are explained again here for clarity. The warmup phase involves standard training of the full supernet (for path-based DNAS) or seed network (for mask-based DNAS). Only the standard weights W are updated during this phase, while NAS parameters θ remain fixed at their initial value (1.0). This ensures that all paths and

Algorithm 2

1: **for** [**do**warmup loop]$e \leftarrow 1,\dots,\text{Epochs}_{\text{wu}}$
2: Update W based on $\nabla_W \mathcal{L}(W)$
3: **end for**
4: $e \leftarrow 1$
5: **while** [**do**search loop] $e < \text{Epochs}_{\text{sr}}$ **or** not converged
6: Update W, θ based on $\nabla_{W,\theta}(\mathcal{L}(W;\theta) + \sum_{j=0}^{J} \lambda_j \max(0, \mathcal{R}_j(\theta) - T_j)$
7: Schedule λ_j, $\forall j$
8: $e \leftarrow e+1$
9: **end while**
10: **for** [**do**fine-tuning loop]$e \leftarrow 1,\dots,\text{Epochs}_{\text{ft}}$
11: Update W based on $\nabla_W \mathcal{L}(W)$
12: **end for**

channels are sufficiently trained. In particular, for path-based DNAS, this leads to a uniform sampling of all supernet paths, while for mask-based DNAS, all channels in the seed model remain active. During warmup, gradients are computed solely with respect to the task-specific loss function $\mathcal{L}$. Since warmup results are independent of constraints, they can be stored and reused across multiple searches (e.g., targeting different hardware configurations).

The second phase is dedicated to architecture optimization. Here, the weights W and the NAS parameters θ are jointly optimized to minimize a specific instance of the DUCCIO objective from Eq. 5.4. This phase runs for a minimum number of epochs ($Epochs_{sr}$) and continues until convergence, utilizing an early-stopping mechanism that monitors the validation loss $\mathcal{L}$. The search terminates once the validation loss ceases to improve. If a predefined validation set is unavailable, we generate one by randomly sampling 10% of the training set.

Upon completing the search phase, we export the final model based on the learned θ values, as described in Sec. 5.2.1 and 5.2.2. We then enter the fine-tuning phase, where, similar to warmup, only the weights W are trained, focusing on minimizing the task-specific loss $\mathcal{L}$. The number of epochs for warmup and fine-tuning is the same as in the MLPerf Tiny official repository (see Sec . 5.3.1 for details).

The rest of this section discusses two critical aspects that make the procedure in Algorithm 2 effective for one-shot DNN optimization under multiple constraints.

Discretized Sampling

After a DNAS search, regardless of the method employed, a single discrete DNN architecture is selected from the solution space and returned. The chosen architecture must satisfy $R_j(\theta) < T_j, \forall j$ in a multi-constraint scenario. To enforce this, we argue against continuous relaxation of the sampling process during the search, as is done in [79] and many other popular DNAS approaches.

To illustrate, consider a simple example: a DNN with one NAS target layer, for which a maximum storage size of $T_j = 600$ is imposed. A path-based DNAS might provide two candidate alternatives for this layer, with sizes $S_0^{(1)} = 100$ and $S_1^{(1)} = 800$, respectively. In the original formulation of [79], these alternatives are combined linearly as in Eq. 5.7 using continuous NAS parameters derived from a standard SoftMax operation. For instance, if at the end of a search loop the optimization yields $g(\theta_0^{(n)}) = 0.49$ and $g(\theta_1^{(n)}) = 0.51$, the estimated size becomes $S^{(n)}(\theta) = 0.49 \cdot 100 + 0.51 \cdot 800 \approx 450 < T_j$. Although this solution meets the constraints within the continuous relaxation framework, i.e., being optimal with respect to the cost component of the loss function, the monotonic nature of the SoftMax implies that the largest θ value determines the final architecture. As a result, the DNAS would ultimately select the second alternative, with a size of 800, which violates the constraint.

In contrast, DUCCIO resolves this issue by performing discrete sampling, that is, by selecting a single concrete architecture during each training iteration. For path-based DNAS, this is implemented via the one-hot function in Eq. 5.2, while in the mask-based approach it is achieved using the Heaviside step in Eq. 5.3. It is important to note that this method differs from those in the original papers—such as [34], which used continuous relaxation for cost (size or OPs) estimation. Other works, including [35, 135], also employ discretized sampling, demonstrating that this approach enhances the correlation between the model's accuracy during search and after final export.

Alternatively, to reduce the gap between the cost estimated during the search and that of the final model in path-based DNAS while retaining continuous relaxation, one can polarize the θ arrays so that their values are well separated. This can be achieved by introducing an additional loss term proportional to the Inverse Coefficient of Variation (ICV) [136]: $\mu(g(\theta))/\sigma(g(\theta))$, where μ and σ denote the mean and standard

deviation, respectively, and $g()$ represents a standard SoftMax or Gumbel-SoftMax function without discretization.

Regularization Strength Scheduling

In our earlier work [125], we demonstrated that enforcing a size target using an absolute value difference formulation can be achieved by applying a large, constant regularization strength λ, thereby eliminating the need for iterative tuning of this parameter. Although this approach remains valid, in the present work, we show that initializing the search with a large λ can lead to suboptimal accuracy, partly due to our novel formulation based on a $\max()$ function. In fact, for specific benchmarks, our empirical findings indicate that the search converges too rapidly toward architectures with costs below the specified constraint, potentially trapping the solution in a suboptimal local minimum with respect to accuracy.

To address this issue, we introduce a scheduling strategy for the λ_j in Eq. 5.4, applying the same schedule uniformly across all constraints. Specifically, we set the target strength as in our previous work:

$$\lambda_{j,target} = \frac{\hat{\mathcal{L}}}{|\hat{\mathcal{R}}_j - T_j|} \tag{5.12}$$

Where $\hat{\mathcal{L}}$ and $\hat{\mathcal{R}}$ denote the task loss and cost measured at the end of the warmup phase. This formulation ensures that, for any cost metric exceeding its target, the corresponding terms in the summation of Eq. 5.4 eventually scale to be comparable in magnitude with the task loss. However, instead of applying this target value immediately, we increase it linearly over a fixed number of initial epochs in the search phase. In practice, at each epoch, we update:

$$\lambda_j = \min\left(\frac{e \cdot \lambda_{j,target}}{\text{Epochs}_{\text{sr}}}, \lambda_{j,target}\right) \tag{5.13}$$

Our experimental results indicate that this scheduling strategy yields more stable performance than a fixed λ_j across different tasks, DNAS methods, and cost targets.

5.3 Experimental Results

5.3.1 Experimental Setup

We developed DUCCIO in Python and PyTorch, leveraging the open-source PLiNIO library [32]. To assess its performance, we selected five edge-relevant tasks, which include the four tasks from the MLPerf Tiny suite [121] in addition to image classification on the Tiny ImageNet [51] dataset. For each task, a reference DNN is chosen, serving as the seed architecture for the mask-based DNAS described in Sec. 5.2.2 and as the template for building the supernet used in the path-based DNAS of Sec. 5.2.1. Specifically, the supernet is constructed by replacing every convolutional layer in the reference DNN with one of four alternatives: i) a convolution with a 3×3 filter, ii) a convolution with a 5×5 filter, iii) a depthwise-separable convolution—that is, a sequence comprising a 3×3 depthwise convolution followed by a 1×1 pointwise convolution [30], and iv) an identity operation. Note that the identity option is available only for layers where the stride equals 1 and $C_{in} = C_{out}$.

For the MLPerf Tiny suite tasks, we evaluated DUCCIO under both global memory and OPs constraints, employing mask-based as well as path-based DNAS for each benchmark—except for the Anomaly Detection task, since its reference model is an FC autoencoder that does not support multiple layer alternatives. Additionally, we utilized Tiny ImageNet to test the layer-wise memory-aware refinement procedure discussed in Sec. 5.2.3 with the mask-based DNAS, as it allows finer-grain memory optimization.

Benchmarks

We consider the ICL and KWS benchmarks with the same setup described in Ch. 4. Additionally, we consider also the other two tasks included in the MLPerf Tiny benchmark [121], i.e., Visual Wake Words (VWW) and Anomaly Detection (AMD).

VWW is a binary classification task designed to determine whether at least one person appears in a given image. This benchmark is derived from the MSCOCO 2014 dataset [53], which contains more than 100,000 RGB images at a resolution of 96x96x3. For this task, the reference network is a MobileNetV1 [30] with a width multiplier of 0.25, trained over 20 epochs.

AMD aims to identify unusual machine operating sounds using the Toy-car subset from the DCASE2020 dataset [54]. The reference model is a simple, fully connected autoencoder trained for 100 epochs. Additionally, we extend our evaluation to a more complex image classification task based on the Tiny ImageNet dataset [51]. This dataset comprises 100,000 RGB images of size 64x64x3, distributed across 200 categories. The benchmark uses a ResNet18 [25] model as the reference architecture, which is trained for 50 epochs.

5.3.2 Global Constraints

Fig. 5.3 depicts the obtained results when applying DUCCIO to both DNAS ap-

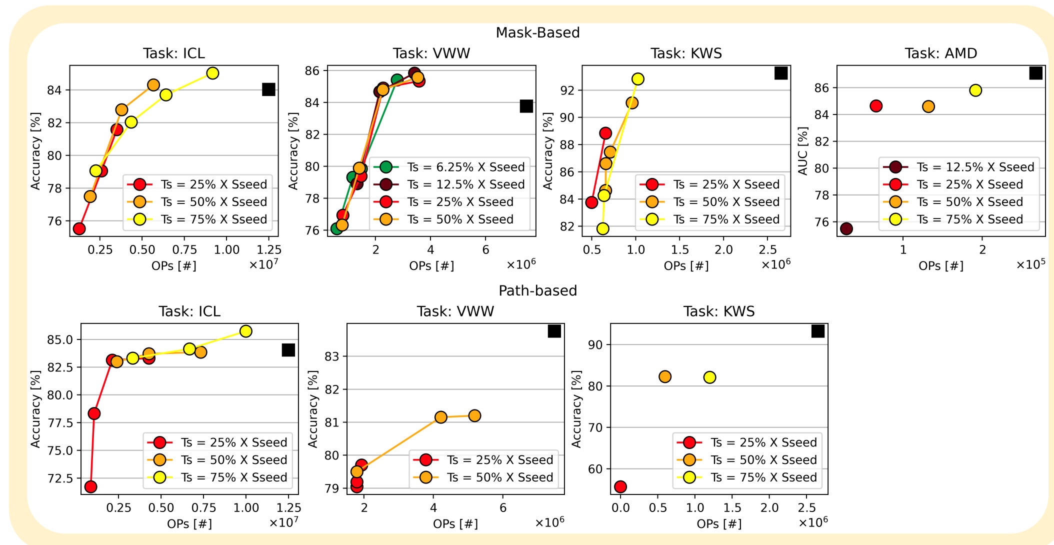

Fig. 5.3 Results for different size targets in the Accuracy versus OPs. (Top row) Results obtained with mask-based DNAS algorithm. (Bottom row) Results obtained using a path-based DNAS algorithm.

proaches on the MLPerf Tiny benchmarks. Each graph displays the reference DNN (black square) alongside the discovered architectures (colored dots) in the Accuracy vs. OPs space, following the global constraints formulation defined in Eq. 5.5. Different colors indicate a different storage constraint, denoted as T_s. We evaluated our method by setting T_s to 25%, 50%, and 75% of the reference architecture's original size. For VWW and AMD, we also tested 6.25% and 12.5% since their reference models are significantly over-parameterized, as highlighted in [125]. This experimental setup replicates real-world deployment scenarios where memory availability

progressively decreases across three different MCU configurations. Within each T_s curve, the rightmost point represents an architecture optimized without any OPs constraint (T_o), while the other points correspond to T_o values set to 25%, 50%, and 75% of the original network's OPs. Notably, each colored dot in Fig. 5.3 results from a single DNAS training run using the specified (T_s, T_o) combination, starting from the seed/supernet.

The graphs on the leftmost side present the results for ICL. Across all memory and OPs constraints, the mask-based DNAS consistently meets the specified targets, covering nearly an order of magnitude in OPs, ranging from 1.07M to 10.0M, while achieving an accuracy between 71.72% and 85.74%. The most significant reduction in OPs while maintaining baseline accuracy (84.31% vs. 84.03%) is achieved using the path-based DNAS under a 50% size constraint and a 75% OPs constraint. This optimized model contains 33.3k parameters and executes 5.68M OPs, representing reductions of 55.9% and 54.6% relative to the baseline, respectively. Additionally, the path-based DNAS yields the highest accuracy (85.7%) while still reducing OPs by 20% compared to the seed network.

The middle-left graphs illustrate the Pareto fronts for VWW. For the mask-based NAS, DUCCIO identifies models with comparable accuracy and latency across all constraints, confirming that even 6.25% of the reference model size is sufficient to reach peak accuracy. The discovered networks span from 602k to 3.42M OPs, with accuracies ranging from 76.1% to 85.83%. Several architectures outperform the seed model in a Pareto sense: the highest accuracy improvement is 2.07% (85.83% vs. 83.76%), while operations and memory are reduced by 54.2% and 87.4%, respectively. For the path-based DNAS, the search space is more constrained, as the reference model predominantly consists of depthwise-separable convolutions. Since our supernets offer only four-layer alternatives, further memory reductions are possible only by replacing layers with identity operations (effectively removing them). However, the results show that layer removal does not benefit this task, as all networks found using the path-based DNAS outperform those generated through the mask-based approach. We omit lower size constraints for this method since they are unattainable within the given search space (the identity layer cannot be chosen when channel counts change or stride > 1).

The middle-right plots present the KWS results. With mask-based DNAS, DUCCIO finds multiple Pareto-optimal architectures for each size constraint. For instance,

under the 75% size constraint and no T_o restriction, the best accuracy achieved is 92.82%, which is only 0.43% lower than the reference model, while reducing OPs by 61.37%. Conversely, the path-based DNAS discovers only three architectures, each derived by replacing one, two, or three of the four depthwise-separable layers in the reference network with identity operations. Once again, the results confirm that the mask-based approach provides superior outcomes, demonstrating that a more granular search leads to better architectures for this task.

The rightmost section of Fig. 3.6 reports results for AMD. As previously mentioned, due to the nature of the reference model, only mask-based DNAS results are presented. Since the model consists entirely of fully connected layers, size and OPs constraints are directly correlated, making separate constraints (T_s and T_o) redundant. As a result, the plot displays single points instead of curves. Nevertheless, DUCCIO successfully identifies Pareto-optimal solutions tailored to different deployment constraints while minimizing accuracy degradation.

The DUCCIO methodology is independent of the specific DNAS approach, making it compatible with slower and faster search algorithms without losing effectiveness. However, since a key objective of our work is to minimize search time, bringing it close to that of a standard training run, it is important to analyze the computational cost of generating the results in Fig. 5.3. As a case study, we measured the average training overhead on the ICL benchmark compared to standard ResNet training. Our experiments were conducted using a single NVIDIA GeForce GTX 1080 Ti GPU, with full reference model training requiring approximately 2 hours. On average, the path-based DNAS incurs a 2.59× time overhead per training step, whereas the mask-based approach increases it by 1.97×. These results are expected, as they stem from the larger size of the supernet in the path-based approach and the additional masking operations in the mask-based method. These findings confirm that DUCCIO remains computationally efficient, even when run on limited hardware resources.

5.3.3 Layer-wise Constraints

This section details the application of DUCCIO to design a network optimized for the memory hierarchy of our target hardware, the GAP8 platform. For this case study, we focus on Tiny ImageNet, as the ResNet18 reference architecture is sufficiently

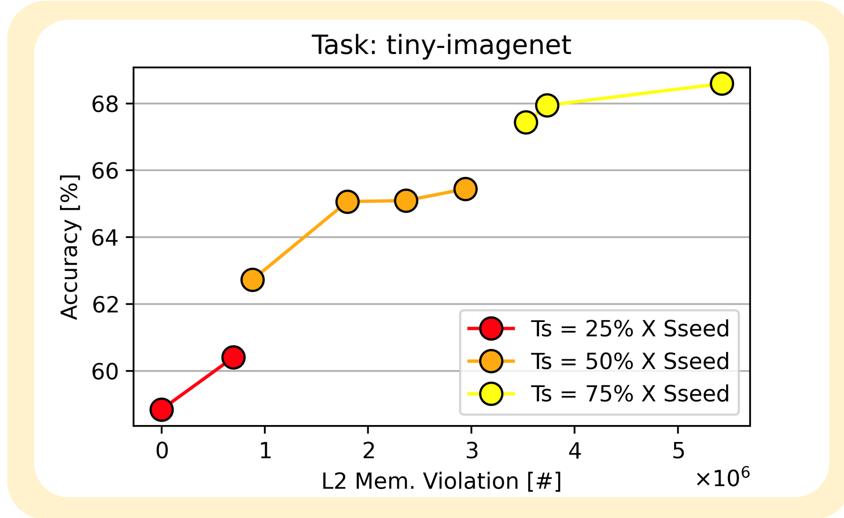

Fig. 5.4 Accuracy vs L2 memory violation.

large to illustrate the impact of layer-wise memory constraints. We employ the mask-based NAS approach, which provides fine-grained control over individual layer sizes.

The seed network consists of 11.28M parameters and achieves a top-1 accuracy of 71.97%. Initially, we search for three smaller models with T_s values set to 25%, 50%, and 75% of the seed (corresponding to the rightmost points in the red, orange, and yellow curves in Fig. 5.4), aligning with scenarios where external L3 memory can accommodate 3M, 6M, and 9M parameters, respectively. Next, we progressively impose stricter constraints on total layer-wise memory usage (Eq. 5.10). Specifically, we define as critical all layers where $\mathcal{M}^{(n)} < (1+C)T_m$, where T_m represents the available L2 memory, and C takes values from $\{0.3, 0.6, 0.9\}$. This process generates the Pareto frontiers shown in Fig. 5.4, where the x-axis represents the total L2 memory violation, calculated as $L2V = \sum_{n \in layers}(\mathcal{M}^{(n)}(\theta) - T_m)$.

Two key insights emerge from this analysis. First, increasing C expands the set of critical layers, thereby reducing L2V. Second, for the same overall model size constraint (e.g., the orange curve), we achieve lower L2V without significantly impacting accuracy. For instance, under a 50% size constraint, we observe that reducing C to 0.3 leads to a 38.9% decrease in L2V while incurring only a 0.38% accuracy drop compared to the rightmost network.

In the following deployment section, we demonstrate that networks with lower L2V generally achieve lower latency and greater efficiency. However, this correlation

is not absolute, as efficiency also depends on (i) the number of layers exceeding the constraint and (ii) the nature of these layers, which influences execution efficiency. For example, depthwise convolutions tend to be less efficient on GAP8 compared to standard convolutions [102].

5.3.4 Embedded Deployment

Table 5.1 Deployment of ICL models on GAP8.

Model	Accuracy	Drop w.r.t Ref	MOPs Constraint	Memory [kB]	MOPs	Latency [ms]	GOPs/s
Reference	84.0 %	0 %	None	75.5	12.50	21.42	0.58
Mask-Based-L	85.0 %	+1.0 %	None	43.4	9.18	21.26	0.43
Mask-Based-L	83.7 %	-0.3 %	6.88	35.6	6.42	15.10	0.43
Mask-Based-M	84.3 %	+0.3 %	6.05	32.5	5.68	15.51	0.37
Mask-Based-M	82.8 %	-1.2 %	4.03	24.7	3.80	10.68	0.36
Mask-Based-S	81.4 %	-2.6 %	None	18.7	5.21	11.66	0.45
Mask-Based-S	79.0 %	-5.0 %	2.61	17.6	2.60	9.40	0.28
Path-Based-L	85.7 %	+1.7 %	10.0	54.8	10.0	17.50	0.57
Path-Based-L	84.1 %	+0.1 %	6.68	51.8	7.55	21.76	0.35
Path-Based-M	83.8 %	-0.2 %	8.19	28.2	7.53	17.56	0.43
Path-Based-M	83.7 %	-0.3 %	5.46	16.4	4.42	14.11	0.31
Path-Based-S	83.3 %	-0.7 %	None	16.4	4.42	14.11	0.31
Path-Based-S	83.1 %	-0.9 %	2.14	14.6	2.48	12.59	0.20

Table 5.2 Deployment after layer-wise memory reduction of Tiny ImageNet models on GAP8.

Model	Accuracy	MOPs	MCycles	GOPs/s	L2V Constraint	Mem. [kB]	L2 Violation [kB]
TinyI-L	68.59 %	1956.8	1002.6	0.19	None	7770	5301
TinyI-L-60	67.94 %	1361.2	644.4 (-36%)	0.21 (+8%)	60 % × L2	6765	3648
TinyI-L-90	67.43 %	1344.3	601.1 (-40%)	0.21 (+15%)	90 % × L2	6615	3446
TinyI-M	65.44 %	1607.6	637.3	0.25	None	5342	2875
TinyI-M-30	65.09 %	1368.1	293.9 (-54%)	0.47 (+84%)	30 % × L2	4932	2313
TinyI-M-60	65.06 %	1164.8	291.5 (-55%)	0.40 (+58%)	60 % × L2	4706	1758
TinyI-M-90	62.72 %	1136.0	252.1 (-61%)	0.45 (+79%)	90 % × L2	3742	861
TinyI-S	60.40 %	1022.3	334.5	0.31	None	2738	681
TinyI-S-60	58.84 %	731.0	178.8 (-46%)	0.41 (+34%)	60 % × L2	2264	0

Table 5.1 and Table 5.2 provide an overview of the deployment results on the GAP8 SoC (ref. Sec. 2.3.2) for the ICL and Tiny ImageNet benchmarks.

Table 5.1 presents the ICL models derived with DUCCIO using both DNAS methods. For each method, two DNNs are reported for each size target, i.e., 75% (denoted with a -H suffix), 50% (-M), and 25% (-L). Precisely, we deploy the two rightmost points from each Pareto front in Fig. 5.3, corresponding to the Mega OPs (MOPs) constraints specified in the table. For reference, the baseline DNN is also included. The “Mem." column indicates the memory footprint of each

model. It is important to note that all DNNs meet the prescribed memory and MOPs constraints. Regarding latency, the mask-based DNAS solutions range from 9.40 ms to 21.26 ms, while those from the path-based DNAS span from 12.59 ms to 27.86 ms. Moreover, within the same size constraint, a higher OPs count generally leads to increased latency, which confirms that OPs serve as a reliable proxy for latency when comparing networks with similar topologies and layer types. The sole exception is the first Path-Based-L model, which, despite having more MOPs than its counterpart, is 4.26 ms faster. This is a consequence of the second model including more depthwise layers. Such layers are characterized by a lower arithmetic intensity, which is a factor not captured by the OPs metric.

Compared to the reference model, two notable “extreme" solutions emerge. The first, obtained via the path-based DNAS (-L), boosts accuracy by 1.7% while reducing latency by 1.22× and memory by 27%. Conversely, the second, produced with mask-based DNAS (-S), sacrifices 5% accuracy in exchange for a 2.28× reduction in latency and a 76.7% decrease in memory footprint. These examples illustrate DUCCIO’s flexibility in maximizing accuracy or compromising it to meet stricter constraints based on the designer’s objectives. Additionally, except for the case above, almost all deployed architectures incur an accuracy drop of less than 2.6% while achieving latency improvements of up to 2.01×.

Table 5.2 details the deployment results for networks on Tiny ImageNet using DUCCIO’s layer-wise size refinement strategy. All DNNs from Fig. 5.4 have been deployed. For each network, this refinement strategy enhances efficiency—measured in GOPs/s—by reducing costly L3 accesses, with improvements ranging from +8% to +84%. As anticipated, the L2V metric correlates with efficiency, albeit not perfectly, due to previously mentioned factors. Furthermore, imposing a tighter constraint (by considering a larger set of layers as critical) consistently leads to latency improvements, as evidenced by the MCycles column, which shows reductions between 36% and 61%. Notably, the TinyI-M-30 model requires 54% fewer cycles than TinyI-M, achieves an 84% increase in GMAC/s efficiency, and suffers only a negligible accuracy loss of 0.35%.

5.3.5 Ablation Studies

Constraint vs Objective

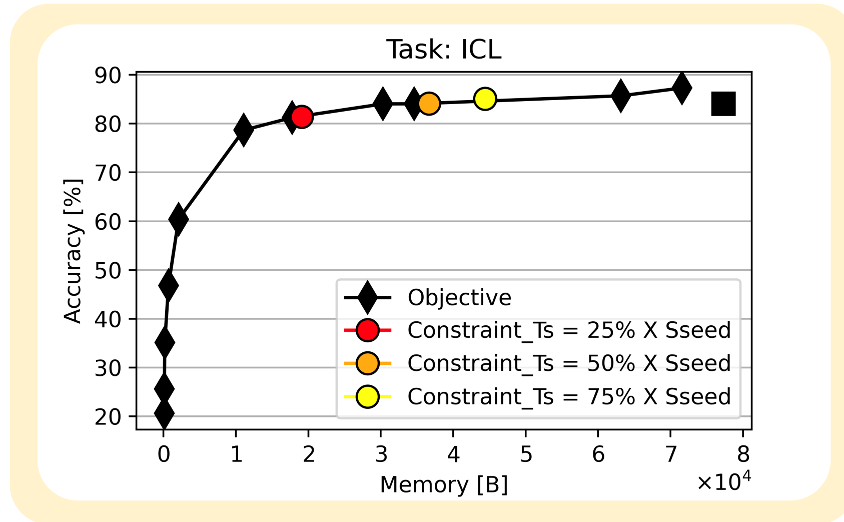

Fig. 5.5 Model size as objective (classical) vs. constraint (DUCCIO).

Fig. 5.5 highlights the key advantage of DUCCIO, i.e., the ability to discover a DNN with specific characteristics in a single run. For this experiment, we executed the mask-based NAS on ICL using DUCCIO's and the traditional DNAS formulation from Eq. 2.11. In each case, only the model size was considered the cost metric. With DUCCIO, we applied the same size targets as detailed in Sec. 5.3.2 while omitting any OPs constraint, so the results correspond to the rightmost points on the curves in Fig. 5.3. Using the classical formulation with twelve different values of λ, we obtained a Pareto curve spanning nearly an order of magnitude in memory usage, with accuracies ranging from 20.64% to 87.25%. This indicates that finding an architecture meeting a specific memory constraint is non-trivial and requires multiple iterations of the DNAS search. In contrast, DUCCIO finds each of the three target points in a single search by simply setting a different memory target T_S. The resulting networks satisfy the imposed constraints and are Pareto optimal, as they lie on the curve obtained by sweeping λ in the conventional DNAS approach. This demonstrates that our formulation preserves the quality of results compared to the classic DNAS method.

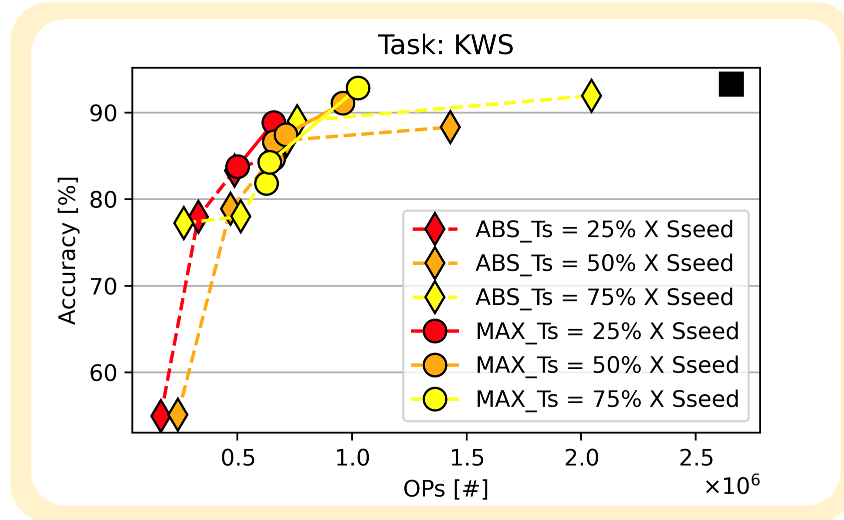

Fig. 5.6 Comparison for different size targets of absolute value and max() constraints.

Max vs Absolute Value

Fig. 5.6 underscores the benefit of using the max function in the loss formulation over the formulation base on the absolute value difference from the target, as explained in Sec. 5.2.3. This experiment is carried out on the KWS task using the mask-based NAS since the reference network is over-parametrized in the number of channels, amplifying the difference between the two formulations. In the figure, the circles represent the results from Fig. 5.3, whereas the diamonds correspond to the results obtained when replacing the max function with absolute values in Eq. 5.5.

Overall, we observe that the architectures derived using the max function consistently Pareto-dominate those obtained with the absolute value formulation for accuracies above 80%. This trend is especially pronounced in the two most complex architectures for $T_s = 75\%$ and 50%. Specifically, when employing the max function, DUCCIO identifies two DNNs with accuracies of 91.06% and 92.82% and with 9.4k and 9.96k parameters, respectively. In contrast, using the absolute value formulation under the same T_s constraints produces models with 88.30% and 91.92% accuracy and 12.1k and 16.7k parameters.

These results indicate that forcing the DNAS to generate a model that closely aligns with the target by using the absolute value difference leads to networks that remain overly parameterized and generalize less effectively compared to those produced with the max function.

Increasing vs Constant λ

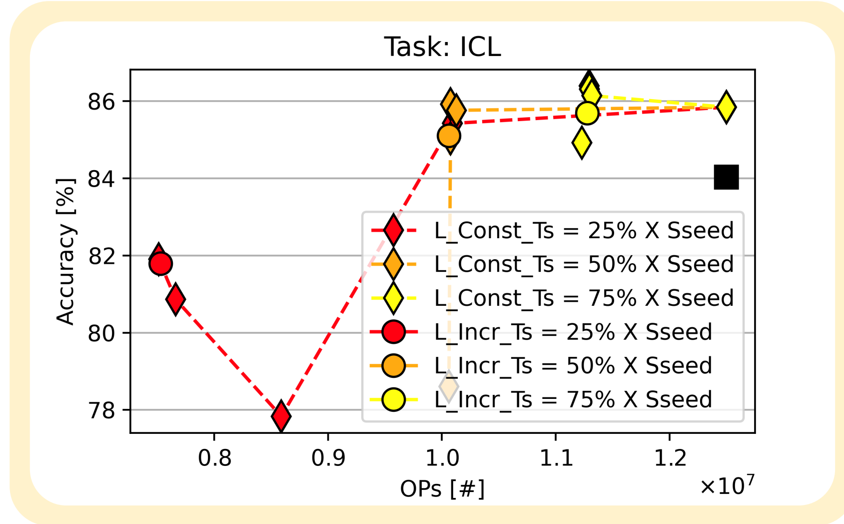

Fig. 5.7 Comparison between the scheduling of λ proposed in Sec. 5.2.4 and a constant λ for DUCCIO's training.

In Fig. 5.7, we demonstrate that the λ scheduling strategy introduced in Sec. 5.2.4 outperforms the use of a fixed regularization strength. To evaluate this, we conducted a search on ICL using the mask-based DNAS with DUCCIO, experimenting with six different constant λ values both below and above the λ_{target} specified in Eq. 5.12. We again set T_s to 25%, 50%, and 75% of the seed size without applying any OPs constraint. Two key behaviors emerge from our findings. First, when the constant λ is too low, the imposed constraint is not enforced, resulting in networks that do not shrink in size (as observed in the yellow and orange rightmost points). Second, when λ is excessively high, the network shrinks too rapidly, removing channels that are important for maintaining accuracy. This effect is particularly evident in the $T_s = 50\%$ case, where the model produced with the largest λ_{const} suffers a 6.5% drop in accuracy compared to the one obtained using the scheduled λ. Importantly, across all T_s settings, the networks derived with the scheduled λ consistently meet the constraint while achieving comparable or superior accuracy with any fixed λ.

Discretized Sampling vs θ Polarization

In Fig. 5.8, we justify our final training decision: employing the Gumbel Softmax trick with discretized sampling to select one among the alternative layers in the

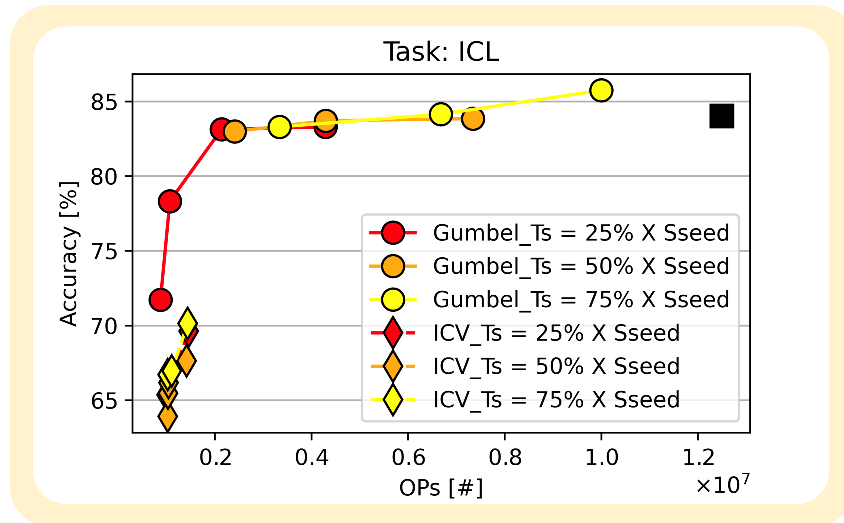

Fig. 5.8 Comparison between discretized Gumbel Softmax sampling and ICV Loss.

path-based DNAS, rather than relying on soft sampling with θ coefficients steered by an ICV loss. This experiment is conducted on ICL with the same targets as those in Fig. 5.3. The difference between these two approaches is clear. The primary drawback of using the ICV loss is that every alternative layer's output still contributes to the overall output of a supernet module—albeit with varying magnitudes due to the polarized coefficients. This leads to weight co-adaptation [74], meaning the weights become adjusted to the fact that the module's output is influenced by all alternatives simultaneously. As a result, when the architecture is discretized at the end of the search, the accuracy suffers significantly, even after subsequent fine-tuning.

Chapter 6

DNN Mapping Optimization on Multi-Accelerator System on Chips

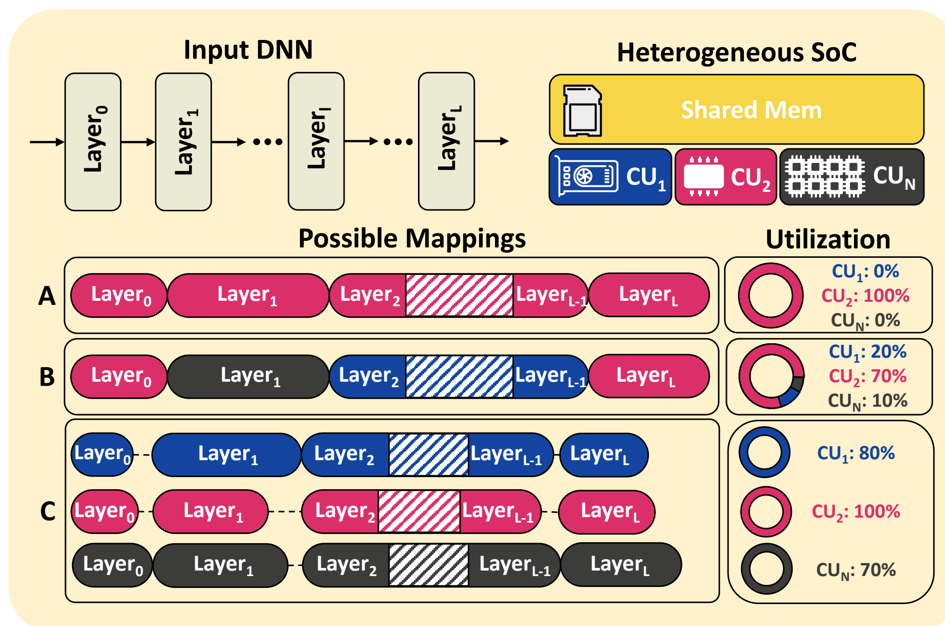

Fig. 6.1 Alternative methods for executing a DNN on a heterogeneous SoC with shared memory. In Strategy A, all computations are carried out on a single CU. Strategy B assigns individual layers to different CUs. Strategy C, implemented in ODiMO, enables fine-grained intra-layer partitioning, where computations within a single layer are distributed across multiple CUs concurrently.

Previous chapters presented hardware-aware software-level techniques to optimize the inference phase of DNNs on resource-constrained edge devices. On the hardware side, the current trend is to achieve highly optimized DNN inference through specialization. This results in the design of heterogeneous SoCs that include multiple specialized Computing Units (CUs) tailored to the execution of specific DNN workloads [137–139, 5, 6]. Execution optimization of DNN models on multi-CU systems presents a substantial challenge. Conventional methods involve executing the entirety of a network on a single CU, as illustrated in Fig. 6.1 (Mapping A). Recent publications have explored multi-CU inference [140–142, 138, 143, 144] through layer-wise partitioning (Mapping B). However, layer-wise partitioning results in suboptimal hardware utilization, given that standard sequential DNNs employ only one CU at any given time. Finer-grained partitioning, wherein each layer is divided among multiple CUs (Mapping C of Fig. 6.1), has been primarily studied for homogeneous CU systems (e.g., multiple GPUs/TPUs) [145, 146], and remains under-explored for heterogeneous systems. Additionally, these works typically assume that *all CUs are capable of executing all DNN layers* and producing equally accurate results. This assumption does not align with numerous real-world scenarios [5, 139, 6]. For example, SoCs incorporating both Digital and AIMC CUs [139, 5] demonstrate this discrepancy. AIMC offers increased speed and energy efficiency but generates approximated results due to extreme weight quantization, such as binary or ternary levels. Conversely, digital CUs, while slower and less energy-efficient, process data with higher numerical precision, yielding more accurate results. Other SoCs [6, 147] feature CUs that are restricted to executing specific DNN layers, like depthwise convolutions. Utilizing these CUs necessitates constraining the DNN to particular architectural patterns, which can influence accuracy, often in exchange for improved efficiency.

This chapter presents **O**ne-shot **Di**fferentiable **M**apping **O**ptimizer (**ODiMO**) a novel approach to optimize and map DNN execution onto heterogeneous systems. ODiMO employs a *training-time optimization* process to explore different mapping alternatives. This optimization process is guided by functional (task performance) and non-functional (energy efficiency, latency) CU characteristics to identify precise and efficient mappings. As the other techniques described in this manuscript, ODiMO employs a gradient-based search method during training, which allows the exploration of fine-grained mappings by segmenting each DNN layer into sub-layers. These sub-layers are then executed concurrently across multiple CUs, as shown in

Mapping C of Fig. 6.1. By accounting for potential task performance reductions due to quantization or layer type selection (e.g., normal vs. depthwise convolution), this strategy tries to counterbalance the accuracy-energy/latency trade-off through analytical and differentiable hardware-aware cost models.

This chapter first provides an overview of the related works in Sec. 6.1, highlighting SotA techniques. Then, Sec. 6.2 details the proposed optimization method. Sec. 6.3 presents the experimental validation of the process. Finally, Sec. 6.4 provides some implementation details to use the ODiMO framework with the considered hardware platforms.

The work described in this chapter has been published in [148, 149].

6.1 Related Works

Efficiently allocating computationally intensive workloads across the CUs available in heterogeneous systems presents a significant challenge. While early research addressed general-purpose applications, such as OpenCL programs [150], recent investigations have increasingly focused on the specific requirements of DNN inference. One line of research considers mapping the entire DNN onto a single CU at a time. For instance, [140] examined a mobile SoC featuring CPU, GPU, and NPU components, selecting the fastest available CU for each inference request to enable parallel processing of multiple requests. A similar approach, HDA [151], targets edge inference servers with custom NPUs supporting varying bit quantization (8, 7, and 6-bit). This work introduces a scheduling method that concurrently assigns entire DNN inference tasks to minimize latency. Upon arrival, a new request is directed to the fastest available NPU that meets a predefined, task-specific accuracy threshold, which accounts for the DNN's offline-profiled resilience to quantization. Other studies explore mapping DNNs at the finer granularity of individual layers. [141] investigated layer allocation between CPU and GPU, employing a random forest predictor at runtime. This predictor uses layer hyper-parameters as input features to determine the mapping that minimizes overall inference energy or latency. [142] profiled DNN execution on a system comprising a GPU (NVIDIA Jetson TX2) and an FPGA (Xilinx Artix7). Based on these profiles, they proposed a heuristic strategy where the entire network is typically offloaded to the GPU, except FC layers, which are executed on the FPGA. Further exploration of layer-level partitioning

includes AxoNN [138], which utilized linear programming to analyze energy versus latency trade-offs when offloading DNN sections to either the GPU or NVDLAs on an NVIDIA Jetson AGX Xavier. For the same platform, [143] proposed an alternative mapping scheme focused on enhancing throughput via data parallelism and pipelining across the GPU and NVDLAs. HaX-CoNN [152] employed an SAT solver to optimize the latency of concurrently executing multiple DNNs, mapped layer-wise onto the CUs of an NVIDIA Jetson TX2. Similarly, Omniboost [153] developed a layer-wise mapping scheme based on Monte Carlo Tree Search to maximize throughput for multi-DNN workloads on a platform with multiple CPUs and a GPU. H3M [154] targeted datacenter FPGAs, using an evolutionary strategy to map multiple DNNs at layer granularity, aiming to minimize the energy-delay product. Finally, MaGNAS [155] addressed the per-layer mapping of Graph Neural Networks (GNNs) onto GPUs and NVDLAs, concurrently optimizing the GNN architecture. This joint optimization, driven by an evolutionary algorithm, seeks to maximize task accuracy while minimizing energy consumption or latency.

All the works cited above operate at a coarse granularity, where the fundamental unit assigned to a specific CU is an entire network layer. In contrast, alternative approaches investigate finer-grained *intra-layer* partitioning. AccPar [145], for example, focuses on DNN training optimization within a cluster of Google TPU-v2/v3 devices. It employs dynamic programming to minimize training latency by partitioning computations along axes such as data batches (data parallelism), input channels, or output channels, considering computation performance and communication costs. Map-and-Conquer [156] examines intra-layer mappings on the Jetson AGX Xavier platform when considering inference workloads. This method utilizes an evolutionary algorithm to explore various partitioning schemes at the channel level for CNNs and the head level for vision transformers, searching for an optimal energy-delay product. Partitioning is achieved by ignoring data dependencies between adjacent weight groups. This optimization occurs post-training, guided by the objective function from [157], initially developed for network pruning. Consequently, the impact on model accuracy is only partially addressed. Furthermore, similar to MaGNAS [155], any accuracy effects considered are related solely to the network architecture, not to the specific characteristics of the target CUs.

Our proposed method differs by employing *fine-grained intra-layer mappings* designed to optimize arbitrary cost metrics, such as energy or latency while maintaining full *task performance awareness*. Notably, ODiMO represents, to the best of our

knowledge, the first instance in the literature applying a gradient-descent-based optimization technique to the problem of mapping and partitioning DNN computations across multiple hardware CUs.

6.2 Proposed Method

Building upon the discussion of Sec. 6.1, it is evident that most existing methodologies are designed without considering task performance. These works primarily focus on optimizing the balance between latency and throughput or latency and energy consumption. Consequently, they do not consider heterogeneous platforms, where the CU selection can significantly influence the precision of the final task. Task performance is critical in contemporary edge-oriented platforms, such as the SoCs detailed in [139, 5, 6]. In such architectures, variations in data representations and limitations in supported operations (e.g., depthwise vs standard convolutions) mean that mapping decisions can substantially impact the achievable task performance. As a result, the direct application of current methodologies to these emerging hardware platforms is not straightforward. While HDA [151] acknowledges the potential task performance variations across different CUs, it tackles a distinct problem, i.e., the concurrent serving of multiple inference requests. Conversely, our work aims to optimize a single, unbatched inference, a prevalent scenario in extreme-edge devices typically designed to process incoming data streams with minimal delay, often in real time.

To our knowledge, ODiMO is the first DNN mapping framework specifically developed for SoCs where CU heterogeneity directly affects the DNN task performance. ODiMO explores the trade-off between task performance and latency/energy during the training phase. This exploration enables determining an optimal DNN partitioning strategy through hyper-parameter optimization, encompassing the layer type and the quantization precision.

Fundamentally, our approach arises from the insight that systems featuring heterogeneous CUs with the above constraints, optimizing a DNN's architecture (e.g., layer type or quantization format), and establishing its mapping onto the diverse CUs are inherently intertwined. Specifically, assigning a particular operation or data format to a portion of the DNN intrinsically enforces that this segment will be executed on the specific CU(s) capable of supporting it. This coupling allows us to

adapt established gradient-based DNN optimization techniques to effectively address the challenges of DNN mapping on heterogeneous CUs.

In contrast to conventional approaches that limit mapping strategies to layer-wise assignments at a coarse granularity, i.e., allocating entire layers to single CUs, ODiMO introduces support for fine-grained intra-layer partitioning as it is shown in Fig. 6.1. This granularity leads to improved resource utilization across all available CUs.

The rest of this section is structured as follows. Sec. 6.2.1 defines the network optimization and mapping problem and subsequently introduces the proposed ODiMO strategy. Sec. 6.2.2 details the application of ODiMO in mapping DNNs onto CUs characterized by incompatible quantization formats. Finally, Sec. 6.2.3 presents the mapping case onto SoCs equipped with specialized hardware units.

6.2.1 Mapping optimization strategy

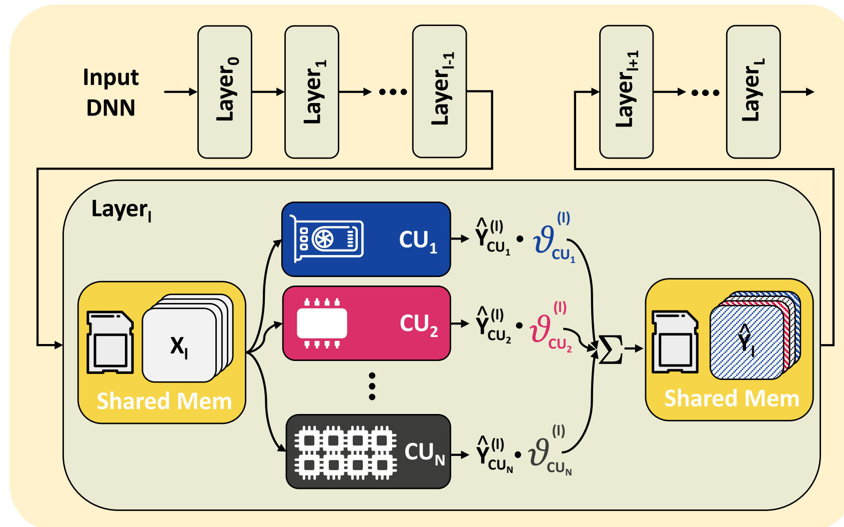

Fig. 6.2 ODiMO mapping strategy of a layer on N different possible CUs.

ODiMO is designed to partition a DNN across a SoC composed of N heterogeneous CUs. Specifically, ODiMO explores the mapping of each Convolutional (Conv) or FC layer at the granularity of *individual output channels/features* among the available CUs. For clarity and without loss of generality, we will consistently use the term “output channels" throughout this discussion. Fig. 6.2 illustrates ODiMO’s mapping strategy for a convolutional layer, where all CUs receive the entire layer’s

input and compute a distinct subset of output activation channels. ODiMO operates under the assumption of heterogeneous systems where all CUs can access a shared memory region (depicted in yellow in Figure 6.2). This shared memory facilitates the loading of layer inputs and the storage of partial outputs. By partitioning computations along output channels, each CU processes a unique set of weights, which may be stored in private memory, and writes its results to dedicated memory locations. It is important to note that while each input activation is loaded N times (once per CU), this apparent data transfer redundancy is accounted for in our optimization's cost models, as detailed subsequently. This approach remains advantageous despite the overhead, as it enables the parallel utilization of multiple CUs.

We assume the shared activation memory is multi-ported, multi-banked, and has sufficient bandwidth to support concurrent access from all CUs. We assume that contention for the same memory location is resolved through an arbitration mechanism. These specifications align with various contemporary hardware designs, including those presented in [145, 5, 139, 6]. Although our current work assumes shared memory, it is worth noting that the proposed methodology could be extended to accommodate CUs with private memories by incorporating the overhead associated with broadcasting input data.

The fine-grained intra-layer mapping strategy implemented by ODiMO introduces a substantial increase in the search space of possible mappings. For instance, with merely $N = 2$ CUs and a ResNet18 CNN, the number of potential channel-to-CU assignments reaches approximately 10^{39}. To manage this huge search space effectively, ODiMO employs a differentiable optimization approach. In this approach, the possible mappings are parameterized by a set of trainable parameters, denoted as θ.

Consequently, by formulating and solving an optimization problem akin to Eq. 2.11, we can efficiently explore diverse mappings *concurrently with DNN training*. This simultaneous process balances computational cost and task performance in a *unified optimization process*.

As depicted in Figure 6.2, ODiMO identifies the DNN layers, denoted by l, that are eligible for mapping onto the available N CUs. Subsequently, for each of these layers, ODiMO simulates the effect of offloading the computation of *the entire layer's output* $\hat{Y}_i^{(l)}$ to *each* CU. This yields a set of N potential outputs, represented as

$\left\{\hat{Y}^{(l)}_{CU_j}\right\}^{N}_{j}$. Finally, the *effective* output feature map $\hat{Y}^{(l)}$ is constructed by computing each of the $C^{(l)}_{out}$ output channels as a linear combination of the outputs generated by each CU. The weights for this combination are determined by the trainable parameters $\theta = \left\{\theta^{(l)}_{CU_j,c}\right\}^{N,C^{(l)}_{out}}_{j,c}$. This method allows ODiMO to produce each layer's output as the composition of the computations performed by all available CUs, reflecting their respective computational characteristics. Mathematically, each output channel c of each layer l is computed as follows:

$$\hat{Y}^{(l)}_{c} = \sum_{j}^{N} \theta^{(l)}_{c,\mathrm{CU}_j} \hat{Y}^{(l)}_{c,\mathrm{CU}_j} \tag{6.1}$$

During optimization, for each channel c in layer l, ODiMO favors assigning channel c to CU j (setting $\theta^{(l)}_{c,\mathrm{CU}_j} = 1$) if this assignment improves the task performance and energy/latency trade-off. Conversely, assignments to other CUs for the same channel (i.e., $\theta^{(l)}_{c,\mathrm{CU}_k}$ where $k \neq j$) are effectively deactivated by setting their corresponding θ parameters to zero. Similarly to what is described in Ch. 4 and Ch. 5, during optimization, the θ values can be treated as discrete samples or relaxed to continuous values within the range [0,1], for example, by applying a softmax operator to a vector of free trainable parameters $\bar{\theta}$. Upon completion of training, the CU associated with the highest sampling probability (or continuous value, in the relaxed case) for $\theta_{CU_{j,c}}$ is designated as the target CU for executing the c-th channel. Following the final channel-to-CU assignments, each layer is restructured into N parallel sub-layers. Each sub-layer comprises the subset of channels assigned to a specific CU j. These N sub-layers can be executed concurrently, and their outputs are then concatenated in the shared memory, serving as input for subsequent layers. Further details on this conversion process are provided in Sec. 6.2.2.

The channel assignment process happens together with the training of the DNN, incorporating the θ parameters as described above and minimizing the loss function defined in Eq.2.11. Specifically, the θ mapping parameters and the standard network weights W are trained in conjunction. The cost term $\mathcal{C}(\theta)$ of the loss function is adaptable and can be configured based on the specific non-functional optimization goal. The framework is flexible and can accommodate any cost metric formulated as a differentiable function of θ. For example, when minimizing latency, ODiMO's

objective function is formulated as:

$$\mathcal{C} = \sum_{l} M^{(l)}, \, M^{(l)} = \max(LAT_1^{(l)}(\theta), ..., LAT_n^{(l)}(\theta)) \tag{6.2}$$

In this formulation, each $LAT_i^{(l)}(\theta)$ represents a differentiable model, parameterized by θ, that estimates the execution latency of the l-th layer on the i-th CU as a function of the assigned channels. $M^{(l)}$ denotes the overall latency of layer l, calculated using a $\max()$ operation. This $\max()$ operation reflects the assumption that all CUs operate in parallel, and minimizing the maximum latency (i.e., minimizing idleness) is a desirable objective for both time and energy efficiency. To ensure the differentiability of the loss term, the non-differentiable max operation in Eq. 6.2 is replaced with a smooth, differentiable approximation. This approximation is computed as the sum of individual latency terms, each weighted by the corresponding softmax-transformed θ parameters. Conversely, for energy reduction, we employ the following cost model:

$$\mathcal{C}_{en} = \sum_{l} (\sum_{i} P_{act,i} \cdot LAT_i^{(l)}(\theta)) + P_{idle} \cdot M^{(l)} \tag{6.3}$$

This model accounts for active and idle power consumption across all DNN layers. For each layer l, the model calculates: (i) the active energy consumption, $\sum_i P_{act,i} \cdot LAT_i^{(l)}(\theta)$, where $P_{act,i}$ represents the average active power consumed by the i-th CU during computation. This is multiplied by $LAT_i^{(l)}(\theta)$, the execution time of the channels assigned to CU i. (ii) The baseline idle energy consumption, $P_{idle} \cdot M^{(l)}$, which captures the platform's idle power P_{idle} over the total layer latency $M^{(l)}$. The total energy cost is aggregated across all layers via the outer summation. The power values, $P_{act,i}$ and P_{idle}, in Eq. 6.3 can be derived from hardware measurements via profiling or obtained from datasheets, depending on the desired level of modeling task performance.

ODiMO's optimization process involves the usual three training phases, already introduced in previous chapters, to generate an optimized mapping, regardless of the specific hardware platform. The first phase, designated as *Warmup*, involves freezing the mapping parameters θ and training the network solely to minimize the task loss $\mathcal{L}$ (without considering the cost term). This is achieved by updating only the standard weights W. Subsequently, the process moves to the *Search* phase. During this phase, both the cost $\mathcal{C}$ and task loss $\mathcal{L}$ are jointly optimized according to Eq. 2.11. The θ parameters and the weights W are updated to identify an effective assignment

of layer partitions to CUs that balances performance and cost. Finally, the *Final Training* phase is executed. In this phase, the channel-to-CU assignments are fixed based on the θ parameter values obtained at the end of the search phase. Like the warmup phase, only the weights W are trained for additional epochs to optimize $\mathcal{L}$ alone. This final training phase aims to mitigate any potential task performance degradation resulting from the discretization of the mapping.

6.2.2 SoC with Incompatible Data Formats

This section describes a practical implementation of the ODiMO methodology tailored for SoCs that integrate multiple CUs with different supported precisions. A representative example of such an architecture is the DIANA [5] presented in Sec. 2.3.4. This SoC incorporates a Digital and an Analog CU, supporting 8-bit and ternary precision for weights, respectively. Other SoCs that comply with this paradigm include the customizable variable-precision NPUs discussed in HDA [151], and the SoC presented in [139], which combines a Non-Volatile Memory Computing CU with an AIMC CU.

In scenarios like these, the challenge of assigning each layer's channels to specific CUs can be reframed as a *precision assignment problem*, similar to the one described in Ch. 4 (without considering pruning). However, it is crucial to note a key distinction: unlike conventional precision assignment, the selection of precision has dual implications here. It not only influences the task performance of the model but also directly impacts inference energy and latency costs. This is because the chosen precision intrinsically constrains the available mapping choices for a given channel to only those CU(s) that support the selected precision level. Consider, for instance, the DIANA SoC. Assigning ternary precision to the c-th channel inherently constrains its execution on the AIMC CU with its associated performance profile. Similarly, designating a channel for 8-bit precision implicitly maps its execution to the digital CU, resulting in a different trade-off regarding latency, energy, and task performance.

Therefore, in the context of the general ODiMO framework outlined in Sec. 6.2.1 (Eq. 6.1), the diverse CU outputs, denoted as $\hat{Y}^{(l)}_{c,\mathrm{CU}_j}$, represent the outcomes of *identical layers but with weights quantized to varying precision levels*. Subsequently, by employing the three-phase training procedure discussed earlier, ODiMO optimizes

the allocation of each channel to a specific quantization bitwidth, and consequently, to the corresponding CU.

It is important to acknowledge a practical consideration. The channel assignments produced by the optimization process are not inherently ordered within each layer. For example, in a layer with $C_{out}^{(l)}$=8 and two CUs, channels 0, 3, and 5 might be assigned to CU_0, while channels 1, 2, 4, 6, and 7 are assigned to CU_1. Direct deployment of the layer in this configuration could lead to implementation inefficiencies due to the interleaved nature of CU outputs in shared memory. A more effective approach involves grouping all channels associated with the same CU into contiguous output memory regions. To this end, Fig. 6.3 illustrates a layer transformation step applied to the DNN post-optimization but pre-deployment onto the target SoC. This transformation is similar to the one discussed in Sec. 4.2.5 and is described again here for clarity. This transformation is depicted for a convolutional layer as a representative example, with activation channels visualized as squares and weight filters as “cubes". The color of the shape outlines signifies assignment to a particular CU. Color patterns are overlaid on specific filters and output slices to enhance clarity. The starting point is the output from ODiMO, shown in the top-left of Fig. 6.3. Subsequently, channels in $Y^{(l)}$ and corresponding filters in $W^{(l)}$ are grouped based on their assigned CU. Furthermore, the weights of the succeeding layer, $W^{(l+1)}$, are also reordered along the *input* channel dimension to maintain the proper network functionality. This rearrangement is illustrated in the central portion of Fig. 6.3. Finally, as depicted in the right section of Fig. 6.3, N independent sub-layers, suitable for parallel execution, are generated by partitioning the original layer. This final step facilitates the deployment of the derived mapping onto the N available CUs without introducing data-marshaling overhead for output aggregation. It is important to note that while the preceding discussion presented the scenario

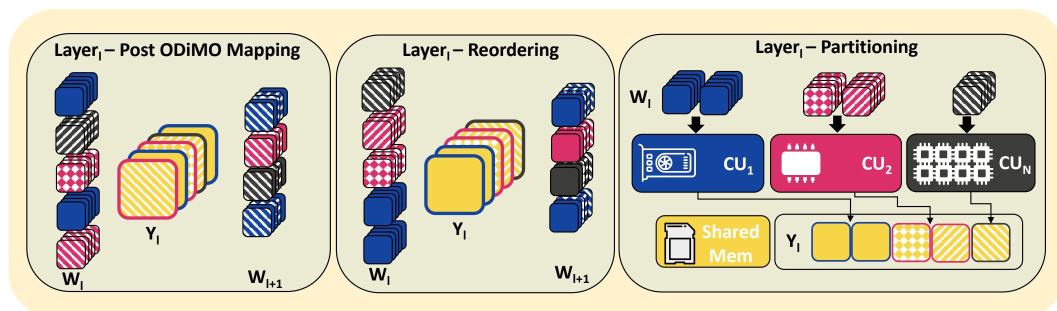

Fig. 6.3 Re-organization pass to enable layer partitioning.

of CUs with incompatible weight quantization formats as a concrete example of the general ODiMO mathematical framework (as defined in Eq. 6.1), our practical implementation diverges slightly for enhanced training efficiency. Although the formulation in Eq. 6.1 remains theoretically sound, we adopt an alternative approach in practice to ease the training process. Specifically, instead of performing a linear combination of the *output activations* produced from each layer instances with varying weight quantization, we exploit the linearity property of Convolutional and FC operations to *directly combine the weights*. This alternative, mathematically equivalent to Eq. 6.1, is expressed through the following factorization:

$$\hat{Y}_c^{(l)} = (\sum_{j}^{N} \theta_{c,\mathrm{CU}_j}^{(l)} W_{c,\mathrm{CU}_j}^{(l)}) * X^{(l)} \tag{6.4}$$

In this formulation, $W_{c,\mathrm{CU}_j}^{(l)}$ denotes the weight kernel corresponding to the c-th output channel of the l-th layer, quantized to the precision level supported by the j-th CU. The term enclosed within the parentheses in Eq. 6.4 effectively constitutes a *composite weight kernel*, representing a weighted average of the kernels from different CUs. The primary advantage of this reformulated approach lies in its computational efficiency. It obviates the need to compute multiple, independent convolution operations for each layer. Instead, it necessitates only the construction of the composite weight kernels, achieved through computationally less demanding element-wise operations such as scalar multiplication and summation. This simplification significantly reduces the overall computational overhead associated with ODiMO training, leading to a more efficient optimization process.

6.2.3 SoC with Specialized HW Units

In this section, we detail the adaptation of ODiMO to heterogeneous SoCs where CUs, while employing compatible data formats, are specialized for executing distinct types of DNN layers. Platforms such as Darkside [6] (Ref. Sec. 2.3.4), featuring a general-purpose multicore RISC-V cluster alongside the dedicated DepthWise Engine (DWE) for efficient depthwise convolutions, exemplify this scenario. Similarly, Kraken [147], a heterogeneous RISC-V SoC incorporating a multicore cluster and a Sparse Neural Engine tailored for event-based spiking DNNs, falls within this category. Conventional DNN architectures often alternate sequentially between

layer types, such as standard and depthwise convolutions. This approach results in a situation where only a single CU is actively utilized at any given time. In contrast, leveraging the ODiMO optimization framework allows for the parallel execution of multiple sub-layers, e.g., simultaneously processing a standard convolution and a depthwise convolution, each operating on a reduced set of output channels. ODiMO optimizes the distribution of output channels across these sub-layers, considering the relative performance characteristics (speed or energy consumption) of the different CUs and the specific task performance implications of each layer type. Analogous to the quantization scenario, the parallel execution of standard and depthwise convolutions introduces a compelling trade-off. While depthwise convolutions involve significantly fewer operations than their standard counterparts, they may impact overall network task performance more pronouncedly.

Following the formulation presented in Eq. 6.1, we again represent the output channels as a weighted combination of CU outputs. In this context, $Y^{(l)}_{c,\text{CU}_j}$ and $Y^{(l)}_{c,\text{CU}_k}$ (where $k \neq j$) represents the results of different tensor operations, each corresponding to the computational capabilities of a specific CU. Although this general framework can accommodate various layer types, including convolutions with differing filter sizes, our practical evaluation primarily focuses on combining standard and depthwise convolutions.

As in the example of Sec. 6.2.2, the direct output of the ODiMO optimization requires a refinement step to produce practically deployable DNNs. Specifically, ensuring that channels assigned to the same CU are located contiguously in memory is essential. If the initial n_c channels are mapped to the depthwise convolution CU, then the subsequent $C_{out} - n_c$ channels should be processed using a standard convolution. As in the quantization case, this constraint is imposed to mitigate costly data marshaling overhead during on-device workload execution. However, unlike the post-optimization layer transformation discussed in Sec. 6.2.2, achieving this channel grouping through a similar post-processing step is not feasible. The inherent structural constraints of depthwise layers, where each output channel is solely dependent on a single input channel, coupled with the potential for reordering enforced by consecutive depthwise layers, preclude the application of a transformation analogous to that depicted in Fig. 6.3. We introduce a constraint directly into the optimization process to guarantee the contiguous grouping of channels assigned to the same CU. Instead of independently applying the softmax function to each element of the θ

array, we employ the following formulation:

$$\theta_i^{(l)} = \sum_{j=1}^{N-i} \hat{\theta}_{N-j}^{(l)} \tag{6.5}$$

This formulation enforces a monotonic non-increasing property: if $i > j$, then $\theta_i^{(l)} \leq \theta_j^{(l)}$. Consequently, this constraint ensures that channels mapped to the same CU are always allocated to contiguous memory regions. Despite this imposed constraint on the optimization, as demonstrated in Sec. 6.3, ODiMO can discover Pareto-optimal mappings that exhibit superior hardware utilization compared to manually designed mappings.

6.3 Experimental Results

6.3.1 Setup

We assessed the performance of ODiMO using three datasets relevant to edge image classification: i) CIFAR-10 [50]; ii) CIFAR-100 [50]; iii) ImageNet-1k [55]. We selected two distinct platforms for our experiments, Diana [5] (Ref. Sec. 2.3.4) and Darkside [6] (Ref. Sec. 2.3.4). These platforms serve as representative examples of SoCs, including CUs with either incompatible data formats (as detailed in Sec. 6.2.2) or support for varying layer types (refer to Sec. 6.2.3), respectively.

A crucial aspect of our methodology that differs for different hardware targets is the formulation of the cost term $\mathcal{C}$ of the optimization objective, as expressed in Eq. 6.2. Furthermore, the Warmup-Search-Final Training protocol, detailed in Sec. 6.2.1, requires slight adjustments depending on the specific hardware. This is particularly true for the Warmup phase. These platform-specific details are described in Sec. 6.4.

For each considered dataset, we utilized DNN architectures well-suited for deployment on either DIANA or Darkside as blueprints for our optimization process. Specifically, for Darkside across all three benchmark datasets, we employed MobileNetV1 [30]. We chose MobileNetV1 as a robust baseline featuring DepthWise-separable convolutions—which can leverage the DWE—and as the foundation upon which to construct the supernet optimized by ODiMO. Specifically, starting from

the original MobileNet architecture, we enabled ODiMO to optimize a supernet incorporating two options (standard Conv and DW Conv) for every layer where $C_{out} = C_{in}$. Consequently, the optimal mappings identified by ODiMO will yield new architectures where both separable and standard convolutions can be executed concurrently. This represents a deviation from the MobileNetV1's sequential execution approach. Conversely, for the DIANA platform, which is less efficient in executing depthwise layers (depthwise convolutions are supported on the digital CU only and exhibit reduced arithmetic intensity compared to standard convolutions), we opted for networks from the ResNet [25] family. In practice, we utilized a ResNet20 for CIFAR-10 and ResNet18 for CIFAR-100 and ImageNet. ODiMO is implemented using Python and PyTorch, building upon the PLiNIO [32] DNN optimization library.

To evaluate ODiMO, we compared it against several baseline mapping strategies, with the baselines varying depending on the HW platform. On DIANA, the baselines are: i) *All-8bit* and *All-Ternary*, representing mappings that exclusively use the digital and AIMC CUs, respectively; ii) *IO-8bit/Backbone Ternary*, a heuristic method from [5] that assigns input and output layers to the 8-bit CU and intermediate layers to the AIMC CU, based on the principle that aggressive quantization near the input/output can negatively impact accuracy; iii) *Min-Cost*, a deterministic optimized mapping that, like ODiMO, uses channel-wise partitioning but focuses solely on minimizing cost, disregarding accuracy. Specifically, it pre-assigns layer channels to the AIMC and digital CUs to minimize Eq. 6.2 or Eq. 6.3. In cases of equivalent costs, digital channels are selected to improve accuracy. Thus, this baseline represents a mapping that optimizes load distribution between the available CUs. Furthermore, in Sec. 6.3.3, we present a comparison of ODiMO solutions against the same PIT [34] structured channel pruning technique of Ch. 3, extended to 2D Conv as described in Ch. 5, using the CIFAR-10 benchmark.

For Darkside, we considered three baseline approaches. The first is the standard MobileNet architecture, which employs Depthwise-Separable convolutions, specifically alternating Depthwise (DW) and pointwise (1x1 filter) convolutions. Secondly, we examined the case where entire layers are mapped to either the cluster or the DWE. When the whole network is mapped to the cluster, we replaced the original Depthwise-Separable convolutions with 3×3 standard convolutions. Conversely, when mapped to the DWE, 3×3 DW convolutions were used. Finally, in Sec. 6.3.3, we demonstrate ODiMO's application to MobileNets initialized with different width

multipliers, and we compare our method's solutions on CIFAR-10 against a standard path-based DNAS approach where whole layers are assigned to a single CU.

6.3.2 Search-Space Exploration

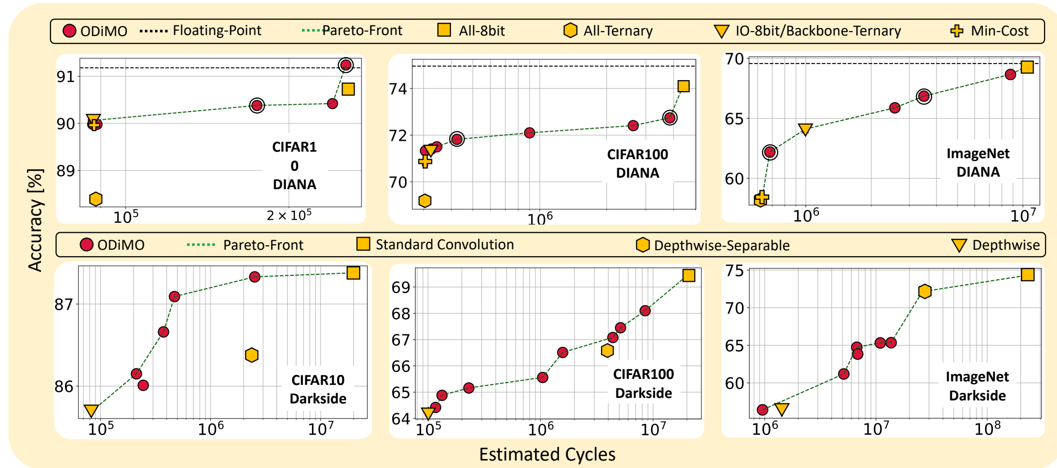

Fig. 6.4 ODiMO mappings discovered when using as optimization target the latency.

Latency Optimization

Fig. 6.4 illustrates the performance of ODiMO across the three benchmark datasets, presenting accuracy against estimated latency for both DIANA (top row) and Darkside (bottom row) platforms. Latency estimations are based on the cost models available in Sec. 6.4. All reported accuracies represent test set performance, with Pareto-optimal points determined using the validation set. Each data point for ODiMO is generated by varying the regularization parameter λ in Eq. 2.11. We have also included the baseline results discussed in Sec. 6.3.1 in yellow, and a horizontal dashed line indicates the accuracy of a floating-point DNN.

Observing the graphs, it is evident that most ODiMO-generated solutions either outperform the baselines or lie on the Pareto frontier. Crucially, ODiMO generates a *diverse collection of Pareto-optimal mappings* that achieve intermediate trade-offs between accuracy and latency, effectively bridging the performance gap between the various baseline approaches. Such intermediate solutions would not be obtainable otherwise. This outcome highlights the effectiveness of our methodology, even

considering the average 1.92× overhead during training. Although training overhead is a one-time expense, even minor enhancements in inference efficiency become significant when scaled to large-scale real-world deployments across numerous devices, especially considering the cumulative number of inferences over a system's lifetime.

Specifically for DIANA and the CIFAR-10 dataset, ODiMO enables a trade-off between estimated latency and accuracy compared to the All-8bit baseline. For instance, a 1.48× speed-up is achieved with a negligible accuracy reduction of less than 0.5% (3rd red dot from the right in the top-left graph). Furthermore, ODiMO identifies a mapping (1st red dot from the right) that matches the floating-point accuracy and surpasses the All-8bit accuracy by +0.5%, demonstrating the regularization benefits of ternarization. In the CIFAR-100 experiments, our method yields solutions spanning an order of magnitude in latency. These solutions provide speed improvements of 1.15× and 4.9× with accuracy losses below $<1.5\%$ and $<2\%$ relative to the 8-bit baseline, respectively (1st and 3rd red dots from the right, top-middle figure). We observe a 0.5% accuracy improvement at equivalent cycle counts compared to the Min-Cost baseline. On the more complex ImageNet task, ODiMO similarly produces a range of Pareto-optimal points spanning more than an order of magnitude in estimated cycles. Specifically, when compared with All-8bit, latency is reduced by up to 1.2× and 3× for accuracy drops of $<1\%$ and $<2.5\%$ (1st and 2nd red points from the right). Moreover, with a 1.08× increase in cycle count, we achieve a +3.8% accuracy gain over the Min-Cost baseline (leftmost red point). This outcome demonstrates ODiMO's capability to discover mappings that effectively balance accuracy and the utilization of available CUs.

The bottom row of Fig. 6.4 displays results for the Darkside platform. For the CIFAR-10 benchmark (bottom-left figure), latency reductions of up to 8× are observed while maintaining accuracy comparable to the standard convolution baseline (rightmost red point in the bottom-left plot). When benchmarked against the Depthwise-Separable baseline (vanilla MobileNetV1), ODiMO achieves a +0.7% accuracy increase alongside a 4.98× speed-up (2nd red point from the right). For CIFAR-100, ODiMO generates Pareto-optimal solutions that cover two orders of magnitude in latency. Notably, with accuracy reductions of -1.5% and -0.1%, speed improvements of 2.4× are achieved relative to standard and depthwise-separable convolutions, respectively (1st and 4th red points from the right in the bottom-middle plot). Finally, the rightmost plot in the bottom row of Figure 6.4 presents

the mappings obtained with ODiMO for the Darkside platform on the ImageNet task. In this scenario, the search space was constrained between the Depthwise and Depthwise-Separable baselines as corner cases. To achieve this, layer alternatives were limited to DW or DW-Separable (DW + Pointwise) Convolutions rather than DW versus standard Conv. This design choice was driven by preliminary experiments indicating that broader search spaces resulted in mappings collapsing toward either the Depthwise or Standard Convolution baselines. Even on this demanding task, Pareto-optimal mappings are obtained. Remarkably, cycle counts are reduced by 1.49× while improving accuracy by 0.2% compared to the Depthwise baseline (leftmost red point). In the higher accuracy range, a 6.8% accuracy drop relative to the Depthwise-Separable baseline is observed, which is partially compensated by a 2.5× reduction in cycles.

Energy Consumption Optimization

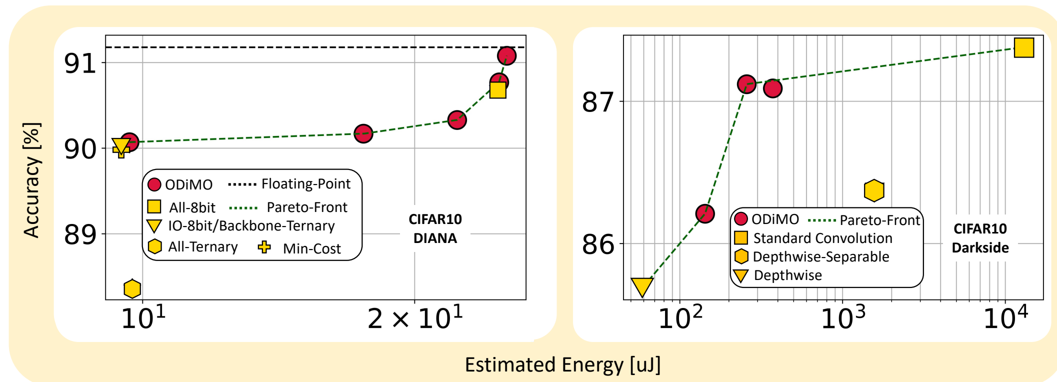

Fig. 6.5 ODiMO mappings discovered when using as optimization target the energy consumption.

To showcase ODiMO's adaptability to different non-functional cost metrics, we developed an energy cost model, formulated as Eq. 6.3, for both the DIANA and Darkside platforms. Eq. 6.3 leverages the existing latency models for each CU on both platforms, integrating them with average power consumption values. The specific mathematical expression of this optimization target is provided as an example in Sec. 6.4.

The results obtained using ODiMO on the CIFAR-10 task, utilizing this energy cost model, are shown in Fig. 6.5. As in the case of latency optimization,

Pareto-optimal mappings are achieved, effectively balancing energy consumption against accuracy for both hardware platforms. Specifically, the results for DIANA are presented in the leftmost plot. Notably, by accepting a minor accuracy reduction of -0.37%, energy consumption can be decreased by 1.41× relative to the All-8bit baseline. Using the Darkside energy model (rightmost plot), energy efficiency improvements of 50.8× and 11× are obtained compared to the Standard and Depthwise Convolution baselines, respectively, with marginal accuracy drops of -0.26% and -0.15%. Furthermore, compared to the Depthwise-Separable baseline, representing the standard MobileNetV1, ODiMO discovered a 6.11× more energy-efficient mapping while enhancing accuracy by 0.76%.

6.3.3 Additional Comparisons

Figure 6.6 compares the ODiMO solutions for CIFAR-10, previously presented

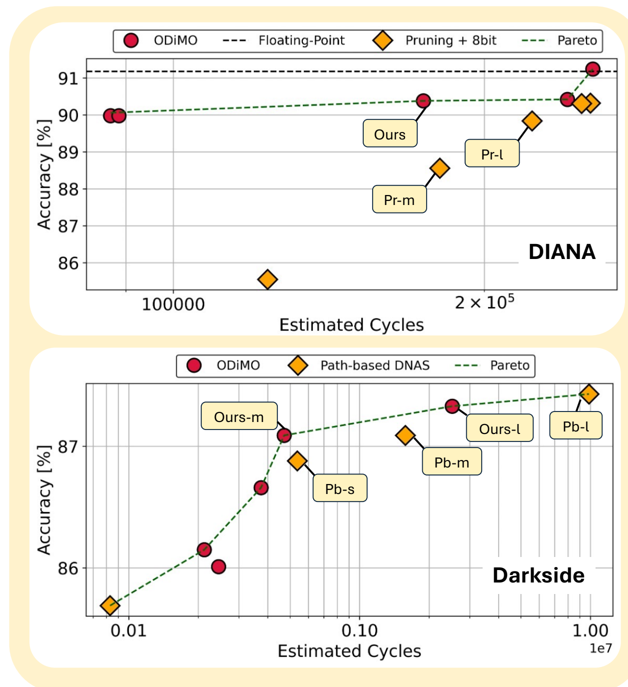

Fig. 6.6 (Top) Evaluation of ODiMO mappings obtained using structured channel pruning and exclusive execution on the digital compute unit of the DIANA platform for the CIFAR-10 task; (Bottom) Evaluation of ODiMO mappings obtained using a layer-wise differentiable neural architecture search strategy on the Darkside platform for the CIFAR-10 task.

in Fig. 6.4, against SotA DNN optimization techniques on both the DIANA and Darkside platforms.

In the first plot, we compare our DIANA results (red circles) with solutions achieved by applying structured channel pruning to the seed network. Subsequently, these pruned networks, quantized to 8-bit, are mapped entirely onto the DIANA Digital CU (orange diamonds). Specifically, we used the PIT pruning methodology detailed in 3 and adapted it for 2D convolutions, utilizing its open-source implementation as in the PLiNIO [32] library. We applied pruning, with varying regularization strengths, to the same network architecture used as the starting point for ODiMO. Notably, all solutions derived using pruning followed by exclusive execution on the digital CU are outperformed by ODiMO. In particular, the ODiMO mapping labeled "Ours" demonstrates a 0.54% higher accuracy and a 21% speed improvement compared to the pruning solution "Pr-l," and a 1.82% accuracy gain with a 3.6% speed-up relative to "Pr-m." This observation reinforces the advantage of mapping less crucial channels in each layer to the less precise yet more efficient Analog CU instead of completely discarding these channels via pruning.

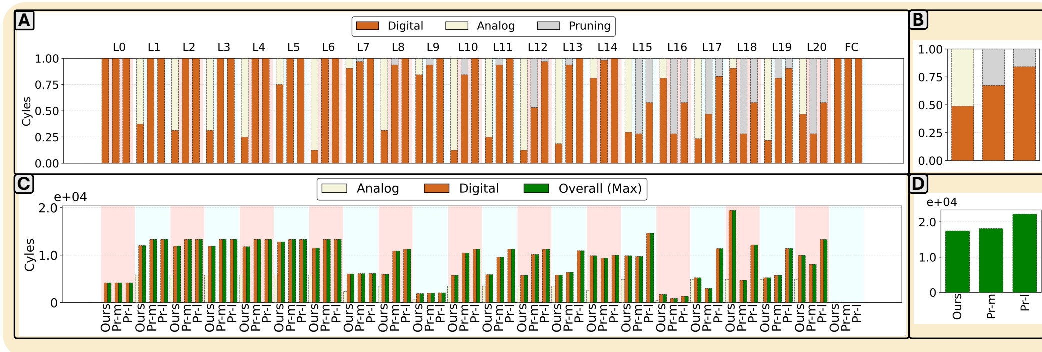

Fig. 6.7 Detailed per-layer comparison of latencies and of the ODiMO assignments for the "Ours" solution and Pruning ratios for the "Pr-l" and "Pr-m" solutions of Fig. 6.6-Top on the DIANA SoC; (A) Layer-level breakdown of CU assignments and channel pruning decisions; (B) Weighted average CU assignment and pruning across entire networks; (C) Per-layer cycle counts for each CU; (D) Total cycle count for each solution.

Fig. 6.7 further explains this finding by detailing the proportion of channels assigned to each DIANA CU, or pruned entirely, across all layers for the "Ours," "Pr-m," and "Pr-l" configurations. It also presents the estimated cycle count for each CU within each layer. For both "Pr" models, pruning is more pronounced in the later layers of the network. In contrast, ODiMO can reduce digital channel usage even in

the initial layers, opting to map them to the Analog CU, which is less precise, rather than directly eliminating them. This approach leads to solutions that are both faster and more accurate. Furthermore, as depicted in Fig. 6.7-B, the “Ours" mapping achieves a nearly balanced 50%-50% utilization of both CUs. Nevertheless, as evidenced in Fig. 6.7-C, the digital CU consistently remains the latency bottleneck, with the analog CU being faster across all layers.

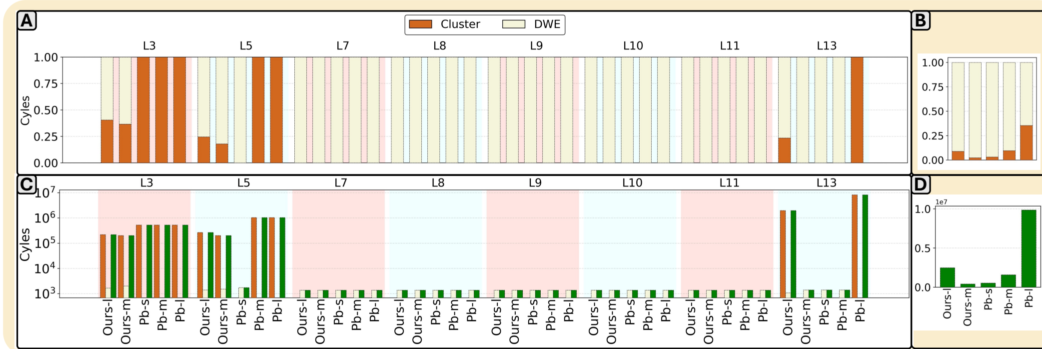

Fig. 6.8 Equivalent of Fig. 6.7 for Fig. 6.6-Bottom on the Darkside SoC.

The second plot in Fig. 6.6 provides a comparison between ODiMO performance on Darkside (red circles) and a path-based DNAS approach, akin to DARTS [79], which performs a coarse-grained selection between layer-level alternatives (orange diamonds). Specifically, we constructed a SuperNet to enable the selection between standard and depthwise convolutions for the identical layers optimized by ODiMO. The layers of the resultant networks are then mapped entirely to one of Darkside's CUs: either the cluster for standard convolutions or the DWE for depthwise convolutions.

In the high-cycle regime, the layer-wise mapping “Pb-l" is on the Pareto frontier and exhibits a marginal accuracy improvement of 0.1% over the ODiMO “Ours-l" solution, but at the cost of a 3.91× increase in cycle count. Within the mid-cycle range, the layer-wise mappings “Pb-m" and “Pb-s" are inferior to ODiMO's “Ours-m." Specifically, “Ours-m" is 3.35× faster than “Pb-m" while maintaining iso-accuracy, and simultaneously achieves a 0.21% accuracy improvement and a 1.14× speed-up over “Pb-s." Moreover, the layer-wise approach fails to discover effective mappings in the low-cycle region, with all solutions converging to the heuristic baseline of mapping all layers to the DWE, as observed in Fig. 6.4. A detailed breakdown of these representative solutions is presented in Fig. 6.8. Referring to Fig. 6.8-A, specific trends emerge across both mapping methodologies. Intermediate

layers are consistently mapped to the DWE, while layers L3 and L5 (proximal to the input) and L13 (near the output) utilize the more versatile standard convolutions on the cluster unit, which operates at a lower speed. This observation aligns with the established understanding that layers adjacent to the input and output are critical for maintaining accuracy. However, while “Pb-l" and “Pb-m" fully map layers L3 and L5 to the cluster, “Ours-l" and “Ours-m," exploiting the ODiMO’s intra-layer mapping capabilities, allocate only a portion of their channels to the cluster. This strategic allocation balances high accuracy with a significant reduction in cycle counts, as illustrated in Fig. 6.8-C and -D. A similar pattern is observed in layer L13, which is fully mapped to the cluster in “Pb-l" but only partially in “Ours-l." These findings underscore ODiMO’s ability to enable novel mapping configurations that achieve an effective equilibrium between accuracy and balanced Compute Unit utilization. Indeed, as evidenced by the breakdowns in Fig. 6.8, employing prior methodologies such as layer-wise mapping leads to reduced CU utilization and reduced accuracy.

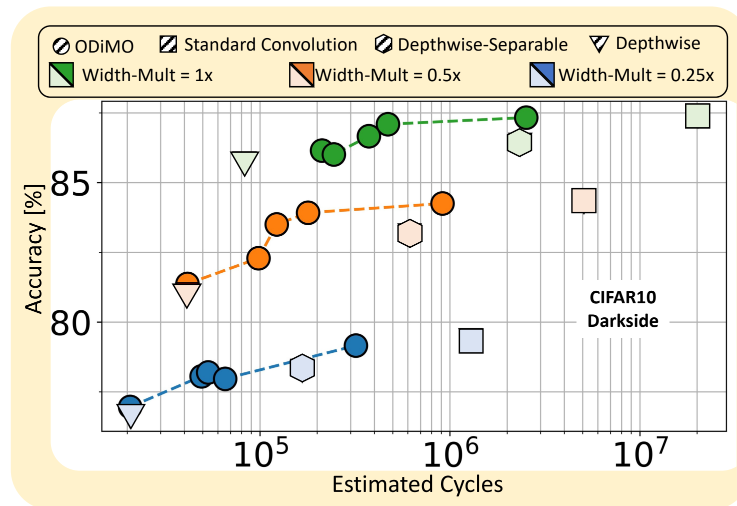

Fig. 6.9 ODiMO mappings obtained using networks with different width multiplier and with latency as optimization target.

Fig. 6.9 presents the mappings derived using ODiMO, on the CIFAR-10 benchmark when using the Darkside latency model. These mappings were generated employing the same MobileNetV1 architecture but with three distinct width multipliers. Specifically, we examined configurations with a width multiplier of $1\times$ (consistent with the DNN used in Fig. 6.4), $0.5\times$ (half the original channels), and $0.25\times$ (a quarter of the initial channels). Across all three of these variations, ODiMO consistently produced a wide set of Pareto-optimal mappings. This result denotes

ODiMO's efficacy, which is irrespective of the channel count, i.e., the geometric dimension of the convolution operation that dictates the partitioning across the heterogeneous CUs. Furthermore, it is noticeable that the mappings identified with a width multiplier of 1× (represented by the green curve) consistently Pareto-dominate the solutions achieved with smaller width multipliers, including the baseline mappings. This phenomenon can be attributed to the efficiency of the DWE, which is capable of processing a substantial number of DW output channels with minimal latency. For example, compared to a standard MobileNetV1 configuration utilizing half or a quarter of the original channels, our approach achieves accuracy improvements of up to 4% and 6.7% at equivalent latency levels. This experiment further corroborates the principle that partitioning the execution of each DNN layer across heterogeneous CUs, enabled by fine-grained mappings, represents a good strategy for latency reduction without the accuracy drops typically associated with channel pruning techniques.

6.3.4 Embedded Deployment

This section describes the validation of the solutions explored in Sec. 6.3.2. Initially, we verify the accuracy of the hardware models for DIANA and Darkside through micro-benchmarking on selected layers. Subsequently, we detail the deployment of a subset of the DNNs discovered by ODiMO on the DIANA SoC. It is important to note that while micro-benchmarking on Darkside was feasible, and we used data from the paper [6] as a reference, the physical hardware is not available to deploy complete networks on the Darkside platform.

Hardware Models Micro-Benchmarking

In this section, we validate the four analytical hardware models, detailed in Sec. 6.4,

Table 6.1 Micro-benchmarking of modeled ResNet and MobileNet layers execution on DIANA and Darkside.

	CU	Error	Spearman	Pearson
DIANA	Digital	42%	95%	88%
	Analog	37%	94%	79%
Darkside	DWE	9%	99.8%	99.9%
	Cluster	16%	96%	98%

developed for DIANA and Darkside. Specifically, we compare the predicted cycle counts against those measured on the actual SoCs for equivalent DNN workloads. These workloads comprise layers extracted from ResNet and MobileNet architectures, encompassing diverse geometries. For each model, we calculated the Pearson and Spearman correlation coefficients, which quantify the linear relationship strength and rank monotonicity between the number of cycles modeled and measured, respectively. The average absolute percentage error between the observed and predicted cycle counts is also reported. Table 6.1 summarizes the outcomes for all four models. While relatively high average errors are exhibited by some models, these primarily arise from unaccounted latency components, resulting in a consistent underestimation of latency. Consequently, the models maintain strong correlations with empirical measurements, as indicated by Spearman coefficients consistently exceeding 94%, making them suitable for utilization within our optimization algorithm. Notably, the models developed for Darkside demonstrate reduced errors and enhanced Pearson/Spearman correlation coefficients compared to those for DIANA. This finding strengthens the validity of the ODiMO mappings obtained for the Darkside platform, even though physical chip unavailability prevents us from deploying entire networks.

DIANA Deployment

This section presents the outcomes of deploying a selected subset of configurations from Fig. 6.4 onto the DIANA SoC, running at 260 MHz. In this evaluation, the previously modeled latency and energy figures are replaced with actual measurements. For each benchmark, we include deployments of the All-8bit and Min-Cost baseline configurations, along with a representative selection of ODiMO-generated solutions (highlighted with black circles in Fig. 6.4). In particular, we choose two Pareto-optimal points—labeled as Accurate and Fast—for every benchmark. Each row in Table 6.2 corresponds to a deployed DNN, for which we report several key metrics: accuracy, energy consumption, inference latency, the utilization percentage of each CU for a full inference pass (*D./A. util.*), and the proportion of output channels executed on the AIMC CU, expressed as the ratio C_{out}^{aimc}/C_{out} across the entire network (*A. Ch.*).

On the CIFAR10 dataset, ODiMO-Fast achieves a 1.45$\times$ latency reduction compared to the All-8bit baseline, with only a minor accuracy drop of -0.32%. This result closely aligns with the 1.48$\times$ reduction predicted by the analytical model (see

Table 6.2 Deployment results of selected solutions from Fig. 6.4 on DIANA.

	Network	Acc.	E. [uJ]	lat. [ms]	D./A. util.	A. Ch.
CIFAR10	All-8bit	90.70	38.70	1.55	100% / 0%	0%
	ODiMO Accurate	91.24	42.50	1.55	100% / 18.6%	5.4%
	ODiMO Fast	90.38	34.44	1.07	100% / 44.8%	51.8%
	Min Cost	90.06	13.6	0.47	9.5% / 93.6%	97.5%
CIFAR100	All-8bit	74.10	756	30.3	100% / 0%	0%
	ODiMO Accurate	72.74	669	26.2	100% / 7.3%	15%
	ODiMO Fast	71.82	65.9	2.14	70% / 58%	96%
	Min Cost	70.86	47.6	1.62	34% / 77.3%	96%
ImageNet	All-8bit	69.33	1578	63.2	100% / 0%	0%
	ODiMO Accurate	66.85	881	33.4	94% / 31%	59%
	ODiMO Fast	62.19	136	4.6	27% / 87%	97%
	Min Cost	58.38	129	4.25	35% / 86%	95%

Sec. 6.3.2), validating its effectiveness. The improvement is obtained by offloading approximately half of the channels to the analog CU. During inference, the digital CU remains active throughout, while the AIMC CU is utilized for 44.8% of the execution time. ODiMO-Accurate, on the other hand, surpasses the All-8bit baseline in accuracy by 0.54% by assigning only a small portion of channels (5.4%) to the analog CU, likely introducing a beneficial regularization effect. Furthermore, the relative ranking of mappings discovered by ODiMO using the analytical models is well preserved during deployment on the actual hardware. Notably, the 1.45× speed-up observed for ODiMO-Fast over ODiMO-Accurate is closely matched by the DIANA latency model, which predicts a 1.46× improvement.

On the CIFAR100 dataset, ODiMO-Fast achieves a 14.2× reduction in latency and an 11.5× improvement in energy efficiency compared to the All-8bit baseline, with an accuracy drop of less than 2.5%. In contrast, ODiMO Accurate keeps the accuracy degradation to just 1.36%, while still being 1.16× faster and 1.13× more energy efficient. When comparing ODiMO Fast to the Min Cost baseline, it delivers a 0.96% gain in accuracy, with a latency and energy overhead of 24.2% and 27.8%, respectively. Although both mappings assign a similar proportion of channels to the analog compute unit (around 96%), the specific channel assignments differ across layers. This variation explains the observed accuracy improvement and leads to a 26% higher median utilization of the digital compute unit.

On the ImageNet dataset, compared to the All-8bit baseline, the ODiMO Accurate configuration achieves a 1.89× speedup and a 1.79× improvement in energy efficiency, with an accuracy drop of less than 2.48%. This gain is obtained by

offloading 59% of the channels to the analog compute unit. The ODiMO Fast configuration enhances the accuracy of the Min Cost heuristic by 3.81%, while incurring only a 1.08× latency and 1.05× energy overhead. Similar to the CIFAR100 case, this improvement comes from assigning a *different* subset of channels to the digital compute unit compared to those selected by the Min Cost heuristic, which does not consider accuracy. Furthermore, the latency impact of this channel reassignment is compensated by a higher overall proportion of channels mapped to the analog CU (97% vs. 95%). These results highlight the importance of the fine-grained, accuracy- and hardware-aware mapping strategy introduced in this chapter.

6.4 Implementation Details

In this section, we detail our proposed method's implementation aspects for the hardware targets, DIANA [5] and Darkside [6]. As discussed in the central sections of this chapter, the two platforms exemplify different architectural challenges: one represents a SoC integrating CUs with incompatible data formats (Sec. 6.2.2), while the other includes CUs equipped with specialized hardware units (Sec. 6.2.3). As such, this section serves as a practical guide on how ODiMO can be adapted to similar heterogeneous targets. For each platform, we first present the analytical latency models employed during training-time mapping optimization (Sec. 6.4.1 and Sec. 6.4.2), followed by a description of how the general ODiMO training protocol was customized (Sec. 6.4.1 and Sec. 6.4.2).

Latency modeling has been a significant focus in recent research on hardware-aware NAS. One prevalent method [35] involves training a compact neural network on a large dataset of measured layers to estimate latency based on layer characteristics. While this technique is compatible with ODiMO, the high predictability of execution on our target CUs allows for a simpler alternative. Specifically, we found that straightforward analytical models, designed to reflect the parallelism and dataflow of the hardware, provide sufficiently accurate latency estimates and significantly speed up the optimization process.

6.4.1 DIANA

DIANA Hardware Models

The DIANA latency models estimate the number of cycles needed to load weights and perform GEMMs or Convolutions, taking into account each compute unit's degree of parallelism and the applied unrolling factors. Non-ideal effects, such as programming overheads and memory access stalls, are not included in the model. Similarly, input loading and output storage latencies are ignored because they can usually be double-buffered with computation. Nonetheless, as experimentally shown in Sec. 6.3.4, this model correlates well with actual latencies measured on the hardware. While the error is, in general, high, both the models for the digital and the analog accelerators neglect the same latency component, therefore showing an almost constant relative underestimation error. The digital accelerator's latency for a Convolutional layer is:

$$LAT^{(l)}_{dig}(\theta) = \lceil \frac{C^{(l)}_{out,dig}(\theta)}{16} \rceil \lceil \frac{o^{(l)}_y}{16} \rceil \times C^{(l)}_{in} \times o^{(l)}_x \times f^{(l)}_x \times f^{(l)}_y + C^{(l)}_{in} \times C^{(l)}_{out,dig}(\theta) \times f^{(l)}_x \times f^{(l)}_y$$

Where $C^{(l)}_{in}$, $o^{(l)}_x/o^{(l)}_y$ and $f^{(l)}_x/f^{(l)}_y$ are the layer's input channels, output spatial dimensions, and kernel sizes respectively. $C^{(l)}_{out,dig}$ is the number of output channels assigned by ODiMO to the digital CU (as a function of the learned θ parameters). The two terms in the latency model correspond to the number of cycles required for MAC operations and DMA transfers, respectively. The MAC-related term accounts for the spatial parallelism of the accelerator, which features a 16×16 array of PEs.

The model for the AIMC CU is:

$$LAT^{(l)}_{aimc}(\theta) = \lceil \frac{C^{(l)}_{in} \times f^{(l)}_x \times f^{(l)}_y}{1152} \rceil \lceil \frac{C^{(l)}_{out,aimc}(\theta)}{512} \rceil \times o^{(l)}_x \times o^{(l)}_y + 2 \times 4 \times C^{(l)}_{in} \times \lceil \frac{C^{(l)}_{out,aimc}(\theta)}{512} \rceil$$

In this case, the first term also models the number of MAC operations as a function of the CU's parallelism ($1152 times 512$), and the second one accounts for DMA cycles.

The two detailed latency models can be used to build the energy consumption optimization target of Eq. 6.3. In particular, we only need the average power consumption of the two CUs when active and idle. Then, for a single layer l, and making the dependency from θ implicit for clarity, the regularization target becomes:

$$\begin{aligned}\mathcal{C}_{en} = &P_{act,dig} \cdot LAT_{dig} + P_{act,ana} \cdot LAT_{ana} \\ &+ P_{idle} \cdot \max(LAT_{dig}, LAT_{ana})\end{aligned}$$

The overall regularization term of the DNN will be obtained by summing a contribution in this form for each layer l.

Training Protocol

This section describes how the general training protocol outlined in Sec. 6.2.1 is explicitly adapted for DIANA. The Warmup phase involves training the provided DNN using floating-point precision. Since DIANA's hardware accelerators do not support Batch Normalization, these layers are fused with the corresponding Conv or Fully Connected layers. To emulate the impact of quantization during training, we adopt the approach proposed in [158]:

$$Q(x) = \frac{e^s}{2^{n-1}-1} \cdot \text{round}(2^{n-1} - 1 \cdot \text{clip}(x, -1, 1)) \tag{6.6}$$

where s is a trainable scaling factor and n represents the bit-width. Regarding Eq. 6.6, we set $n = 8$ for weights targeting the digital accelerator, while $n = 2$ is used to apply ternarization, which corresponds to the quantization format used by DIANA's AIMC accelerator weights. For activations, the digital and AIMC blocks employ slightly different formats—8-bit and 7-bit, respectively. To account for this during the optimization phase, we adopt the more constrained format (7-bit) as the fake-quantization bit-width for all layer inputs and outputs. We found that this approximation did not degrade our results as long as the DNN was appropriately fine-tuned (see below).

Throughout the Search phase, the fake-quantized neural network is trained to convergence using an early stopping criterion. Following this, the final channel allocation to each compute unit is discretized, and the model proceeds to the Final Training stage. During this phase, the precise quantization scheme is applied: shared

activation data are stored in 8-bit format, while the analog in-memory computing (AIMC) accelerator employs 7-bit resolution for its digital-to-analog and analog-to-digital converters, effectively discarding the least significant bit of the inputs and outputs.

6.4.2 Darkside

Darkside Hardware Models

Darkside includes a cluster of eight SIMD-enabled general-purpose RISC-V cores and an HW accelerator, the DWE, to execute Depthwise Convolutions with high arithmetic intensity. The latency model of the DWE is defined as:

$$LAT^{(l)}_{DWE}(\theta) = \lfloor \frac{C^{(l)}_{out,DWE}(\theta)+15}{16} \rfloor \times (o_x o_y \tau_{\text{comp}} + o_x \tau_i + \tau_w)$$

Here, $C^{(l)}_{out,DWE} = C^{(l)}_{in,DWE}$ indicates the number of output and input channels for layer l. The constants τ_{comp}, τ_i, and τ_w are set to 4, 9, and 9, respectively, reflecting the number of cycles required to compute the relevant portion of the output pixels, load a new segment of the input, and fetch the necessary weights. This model only neglects minor latency contributions, such as programming overheads and conflicts in the memory interconnect.

The model of the RISC-V cluster is:

$$LAT^{(l)}_{\text{clust}}(\theta) = \lfloor \frac{o^{(l)}_y+1}{2} \rfloor \lfloor \frac{o^{(l)}_x+7}{8} \rfloor \times \lfloor \frac{C^{(l)}_{out,clust}(\theta)+3}{4} \rfloor (15 + 8 \lfloor \frac{f^{(l)}_x f^{(l)}_y C^{(l)}_{in}+3}{4} \rfloor)$$

The factor within round brackets considers the cycles required to perform matrix multiplications and output activation operations. The factor it multiplies represents the number of iterations required to generate the full output feature map, considering the parallelization across eight cores along the o_x dimension and the inner loop unrolling applied to the o_y dimension with a factor of 2. While the model accurately estimates the cluster performance, the real software implementation, which is not open-source, can slightly deviate from this model.

The energy consumption optimization target for the Darkside platform can be derived starting from Eq. 6.3 by plugging in the analytical latency model as done for the DIANA platform in Sec. 6.4.1.

Training Protocol

The SuperNet ODiMO, employed for optimizing layer choices in the Darkside architecture, provides two alternatives per layer: a standard convolution (Conv) and a depthwise convolution (DW Conv), corresponding to the cluster and DWE compute units, respectively. Before the Warmup phase, both options are initialized with random weights. To guarantee proper training across all paths within the SuperNet, each Warmup iteration independently and uniformly samples a random number of channels from both alternatives at every layer.

In the Search phase, to prevent the optimization process from quickly settling on low-cost solutions with poor performance, we adopted a strength-scheduling strategy akin to the method described in Ch. 5. Specifically, the regularization strength λ is initially reduced by a factor of 100 and then gradually increased in a linear fashion over successive epochs until it reaches its intended value, after which it remains fixed.

After the search phase, the mapping is discretized, and the resulting network architecture is trained from scratch by minimizing only the task loss $\mathcal{L}$ with respect to the weights W. Unlike DIANA, where fine-tuning yielded satisfactory results, we observed that a full retraining from scratch leads to improved performance in this case.

Chapter 7

Applications

This chapter presents some application-oriented works where methodologies detailed in previous parts of this thesis have been employed to optimize the inference phase of DNNs. In this chapter, we consider the case of the design of edge-relevant applications based on complete systems integrating sensors, MCUs, and actuators.

The first application is discussed in Sec. 7.1. It deals with collaborative heart-rate estimation, distributed between a sensor-equipped wearable device with BLE connectivity and a mobile device. A DNN processes PPG and 3D accelerometer signals for robust heart-rate estimation. In this context, the structured pruning approach introduced in Ch. 3 was applied to obtain multiple network variants, each offering different performance-complexity trade-offs suitable for deployment across the two distinct hardware platforms.

Subsequently, a computer vision and robotics application is presented in Sec. 7.2. In particular, a nano-drone [9] utilizes a DNN to estimate the pose of a person within its field of view. The pose information is used to control the drone to follow and maintain the person in its field of view. Also, in this case, the structure pruning technique from Ch. 3, adapted for 2D CNNs as described in Ch. 5, was utilized to discover more compact and computationally efficient DNN models for this task.

Finally, a low-resolution infrared smart camera application for privacy-preserving people counting is detailed in Sec. 7.3. A 2D CNN classifies the collected infrared frames. Also, in this case, the structured pruning approach of Ch. 3 and Ch. 5 is employed. Moreover, given the hardware support, mixed-precision quantization (Ref. 4) is also used.

The work described in this chapter has been published in [29, 159, 8].

7.1 PPG-based Heart Rate Estimation on Wearables

Modern wrist-worn devices are equipped with multiple sensors, including accelerometers for recognizing daily physical activities and evaluating fitness levels, alongside PPG sensors that enable additional health-related features like heart rate monitoring and blood pressure estimation [160]. However, PPG technology, which relies on optical blood flow measurements in capillaries, provides less precise readings than ECGs and is more susceptible to signal noise. Motion Artifacts (MAs) are primary sources of this noise, typically arising from imperfect skin-sensor contact or ambient light leakage during physical movement.

Mitigating the impact of MAs on PPG signals is a challenge addressed by numerous studies. Since introducing the first significant public dataset for PPG-based HR estimation in [161], researchers have developed various algorithms [161, 162]. These often involve filtering techniques or analyzing the correlation between accelerometer and PPG data to counteract MA distortions. While effective under certain conditions, these methods can be computationally intensive and often exhibit limited generalization performance when applied to data from individuals not seen during the algorithm's tuning phase.

Deep learning methods have recently emerged as a promising direction, offering potential improvements in HR estimation accuracy, better generalization across users, and possibilities for reduced computational demands. The release of the PPGDalia dataset [43] (already presented in Sec. 3.3) marked a significant step, providing not only extensive data but also the first deep learning algorithm for this task, based on analyzing the signal's frequency spectrum with 2D convolutional networks. This approach outperformed existing methods but came with substantial computational requirements, making direct deployment on resource-constrained devices difficult. Subsequent research focusing on deployability has mainly explored scenarios where the prediction runs entirely on the smartwatch collecting the data [163] or is offloaded completely to a higher-capability mobile device [164].

This work explores a more flexible set-up where wearable and mobile devices cooperate on HR prediction via a BLE link. To effectively distribute the computational tasks in this two-device system, we introduce CHRIS: a **C**ollaborative **H**eart **R**ate **I**nference **S**ystem. CHRIS is designed to manage a combination of conventional signal processing algorithms and DNNs. CHRIS assesses which available HR tracking

algorithm is most appropriate for each incoming data sample and decides whether it should run on the smartwatch or be sent to the phone, seeking an optimal balance between accuracy and energy usage.

7.1.1 Collaborative Heart Rate Inference System

CHRIS is a lightweight runtime system designed to operate on smartwatches. Its primary goal is to balance HR tracking accuracy and energy efficiency depending on three factors: (i) an estimate of the input's current complexity, (ii) the status of the device's connectivity, and (iii) user-specified constraints related to either accuracy or power consumption. CHRIS manages a collection of different heart rate estimation models to meet these objectives. When a new estimation is required, it selects an appropriate model from this collection and decides whether to execute it locally on the smartwatch or to offload the computation to a connected phone.

Fig. 7.1 depicts the overall framework with its inputs and the resulting network and device selection. The main components are the Models Zoo and the Decision Engine. The Models Zoo provides different models characterized by the energy consumption on both board and smartphone and its task error (Table 7.1). The Decision Engine then chooses a platform and a model to be executed (Fig. 7.2).

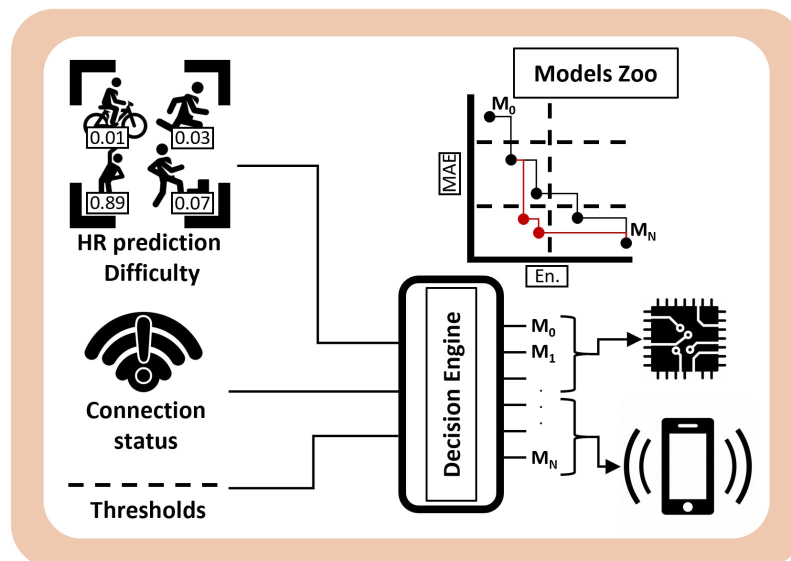

Fig. 7.1 Components of the CHRIS framework.

CHRIS Configurations

Table 7.1 Models used to build CHRIS configurations.

	Energy [mJ]			MAE [BPM]
	Board	Phone	BLE	
AT	0.23	1.61	0.52	10.84
DNN-Small	0.543	5.54	0.52	5.63
DNN-Big	41.11	25.60	0.52	4.88

Table 7.2 Configurations used within CHRIS.

	E. [mJ]	MAE [BPM]	Models	Diff.	Exec.
C_1	0.92	10.11	[AT, DNN-Small]	9	Local
C_2	0.87	10.05	[AT, DNN-Big]	9	Hybrid
...					
C_N	40.05	5.11	[AT, DNN-Big]	1	Local

The first component of CHRIS is a collection of offline profiled *configurations*. The results of such profiling are stored on the smartwatch. A *configuration* is a group of 2 HR prediction models, where for each input window, only one model is selected. Each configuration is characterized by average energy consumption, average MAE, difficulty threshold used by the decision engine (detailed below), and type of execution (entirely on the smartwatch or hybrid). A profiling dataset is used to estimate the average MAE and energy. Each configuration is built with a less accurate but more efficient model and a more accurate but more energy-hungry one.

The characterization of individual models is reported in Table 7.1. Instead, an example of the configurations' profiling information stored in the smartwatch memory is shown in Table 7.2. The model to employ for a given configuration for estimating the HR is selected depending on the *difficulty threshold*. The activities performed by the subjects in the PPGDalia dataset [43] define nine different difficulty levels. These activities, such as sitting, walking, or playing table soccer, induce varying levels of MAs in PPG signals, affecting the difficulty of heart rate HR estimation differently. These activities can be ranked in terms of difficulty based on the average accelerometer signal energy, with lower ranks corresponding to fewer MAs. The CHRIS system uses a configurable *difficulty threshold* to determine which HR estimation model to apply. A simpler and more efficient model is used if an

activity's difficulty is at or below the threshold. Otherwise, a more complex model is selected to handle higher-MA activities.

CHRIS Decision Engine

The internal structure of the decision engine of CHRIS is shown in Fig. 7.2.

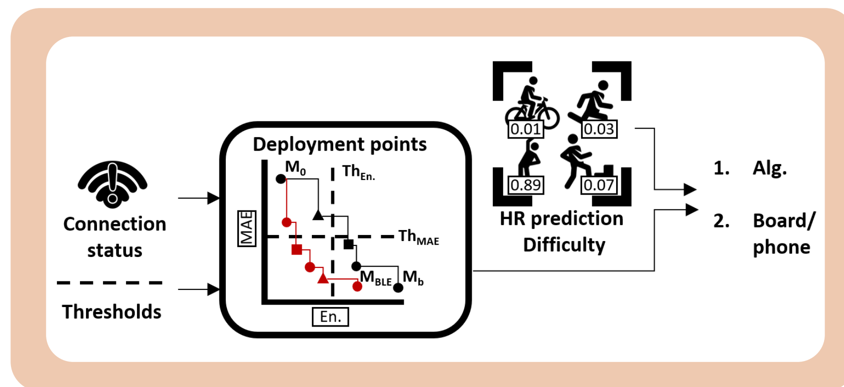

Fig. 7.2 CHRIS Decision Engine

CHRIS selects a suitable configuration from the profiling table stored in the MCU's memory based on the current connection status and a user-specified threshold. The connection status serves to narrow down the search space by excluding *hybrid* configurations—where the larger model runs on the smartphone—if the BLE link is unavailable, thus retaining only *local* configurations, in which both models execute on the smartwatch. The user-defined threshold then guides the final selection within the feasible set. If the threshold is specified as a maximum allowable energy consumption ($Th_{\mathrm{En.}}$, shown as a vertical line in Fig. 7.2), CHRIS chooses the configuration offering the lowest MAE under that energy budget. Conversely, if the threshold is a maximum acceptable MAE (Th_{MAE}, shown as a horizontal line in Fig. 7.2), the configuration with the minimal energy consumption that meets the MAE constraint is selected.

As discussed above, after selecting a configuration (i.e., a pair of HR tracking models) based on the estimated degree of MAs, CHRIS automatically assigns each input window to one of the two models. Accordingly, the decision is based on accelerometer data through a simple activity recognition model that assigns each input window to one of the nine activities described in Sec. 7.1.1. The employed classifier is based on a Random Forest (RF) algorithm. The decision threshold (Ref. Sec. 7.1.1) is compared with the activity performed by the subject given as output by the RF. If the predicted activity's ID is higher than the threshold, the input window

is fed to the most complex model of the pair. Otherwise, the simplest model is executed.

Benchmark HR tracking and activity recognition models

We consider CHRIS configurations that can be obtained using a simple classical algorithm and two DNNs as HR predictors. These three algorithms yield 60 possible configurations when considering three combinations of two out of the three models, 10 difficulty levels, and wearable or mobile as targets for the most complex model. Out of these 60 configurations, the profiling results of the Pareto optimal ones (i.e., 30 configurations) are stored in the MCU. Despite the large number of configurations, CHRIS' memory overhead is limited, given that a smartwatch only needs to store (at most) 3 HR predictors.

The Adaptive Threshold (AT) method described in [165] represents our simplest predictor. It recognizes regions of interest where the raw signal is higher than the mean. Within each region, the highest value is marked as a peak. Then, the HR is computed as the distance between two successive peaks. The cost of AT is limited to $\approx$ 3k operations per input window, but it achieves a far from optimal average MAE of 10.99 BPM on the PPGDalia dataset.

The two DNNs are called DNN-Small and DNN-Big. These are two TCNs obtained using the PIT algorithm of Ch. 3 with the TEMPONet [46] network as seed on the Dalia [43] task. DNN-Small achieves an MAE of 5.6 BPM on Dalia performing 77.63k operations per inference and has 5.09k parameters. DNN-Big (232.6k parameters), instead, obtain a lower MAE of just 4.87 BPM on the same dataset while performing 12.27M operations. The MAEs of these DNNs are then much lower than the MAE achieved by AT algorithm. Nonetheless, this is exchanged with a much higher number of operations ($25.9 times$ and $4090 times$ higher, respectively). Together, these three HR tracking algorithms defines a wide optimization space for CHRIS. Indeed, they span three orders of magnitude in complexity and over 6 BPM of MAE.

Finally, the activity recognition model is a RF with eight trees each with a maximum depth of five. Each tree is fed with mean, energy, standard deviation, and number of peaks extracted from the accelerometer data.

7.1.2 Experimental Results

Setup

The target embedded system for our experiments is the HWatch (Ref. Sec. 2.3.5). We also consider a Raspberry Pi3 equipped with an Arm Cortex-A53 core as a proxy of typical mobile processors. The proprietary ST neural network deployment toolchain X-CUBE-AI [166] is used to deploy the prediction models on the STM32WB55 in the HWatch. Instead, the TensorFlow Lite interpreter is used to execute the models on the Raspberry Pi 3. Before deployment for both target platforms, DNN-Small and Big are quantized to 8-bit using TensorFlow Lite quantization-aware training.

Individual Models Deployment

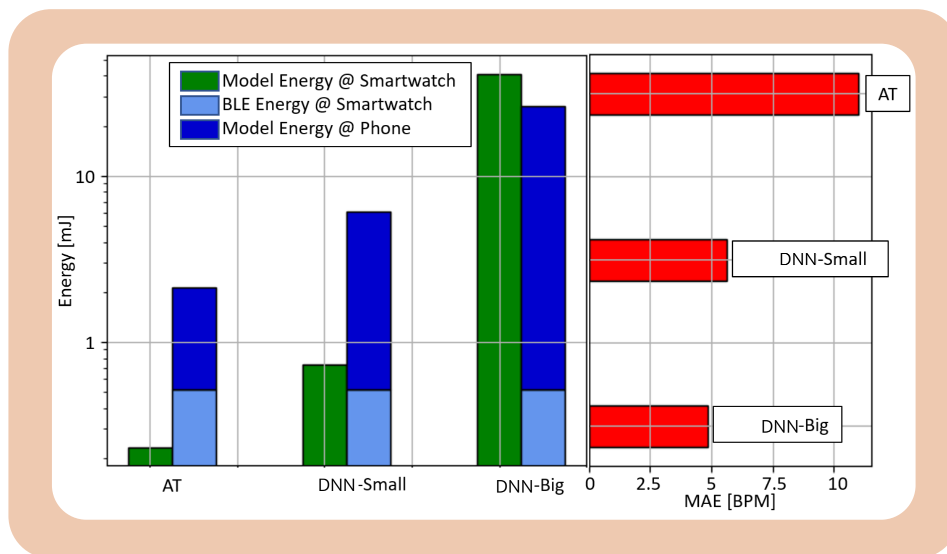

Fig. 7.3 Baseline models energy consumption (left) and average MAE on PPG-Dalia (right).

Table 7.3 Baseline models deployment on the Raspberry Pi3 and on the STM32WB55.

	Raspberry Pi3		STM32WB55			
	Time [ms]	Energy [mJ]	Cycles	Time [ms]	Energy [mJ]	MAE [BPM]
AT	1.00	1.60	100k	1.563	0.234	10.99
DNN-Small	3.45	5.54	1.365M	21.326	0.735	5.60
DNN-Big	15.96	25.60	103.16M	1611.88	41.11	4.87
Bluetooth	n.a.	n.a.	n.a.	10.240	0.52	n.a.

Raspberry Pi3 Frequency = 600 MHz. STM32WB55 MCU Frequency = 64 MHz.

The left-most part of Fig. 7.3 shows the energy required by computations on the mobile device (dark blue), the same energy required on the HWatch MCU (green bar), and the energy of BLE transmission (light blue). This last term is constant, given that the input sample dimension is also fixed. Table 7.3 details these numerical results with the number of clock cycles on the HWatch and the execution time of each model on both platforms.

AT, which consumes 0.234 mJ on the HWatch, is the least accurate but most efficient algorithm. Executing this algorithm on the smartphone is sub-optimal since the energy consumed by the watch for streaming data and the smartphone's CPU would be higher (0.234 mJ vs. 0.519 mJ and 1.604 mJ, respectively).

DNN-Small trade-offs MAE and energy. Indeed, the consumption on the HWatch is $3.1 times$ higher than that of AT, for a 1.96× reduction in MAE (5.6 BPM vs 10.99 BPM). For this reason, it is convenient from the whole system's energy perspective to run HR tracking on the smartwatch with this model. On the other hand, if the objective is to optimize the energy consumed by the smartwatch then offloading the HR tracking to a phone is more convenient (0.519 mJ for BLE transmission vs. 0.735 mJ for execution).

Lastly, the most accurate and energy-hungry model is DNN-Big. This model requires 41.07 mJ per prediction on the HWatch, i.e., two orders of magnitude more energy than AT. For this algorithm, considering both the smartwatch's consumption and the total system energy, the local execution on the smartwatch is always sub-optimal. Indeed, running the model on the Cortex-A consumes 25.60 mJ, while BLE transmission requires the usual 0.519 mJ.

CHRIS Exploration of MAE vs. Energy

Fig. 7.4 shows all the possible configurations CHRIS covers in the space MAE vs. smartwatch energy. This analysis considers smartwatch energy because HR tracking usually represents only a small portion of the overall power budget on mobile devices, which typically manage many tasks concurrently. Within the figure, the green diamonds denote the baseline strategies outlined earlier. Specifically, the point marked *BLE + DNN-Big* represents a scenario where HR processing is always offloaded to the mobile; given that mobile energy optimization is not an optimization factor in this experiment, the most accurate model available (DNN-

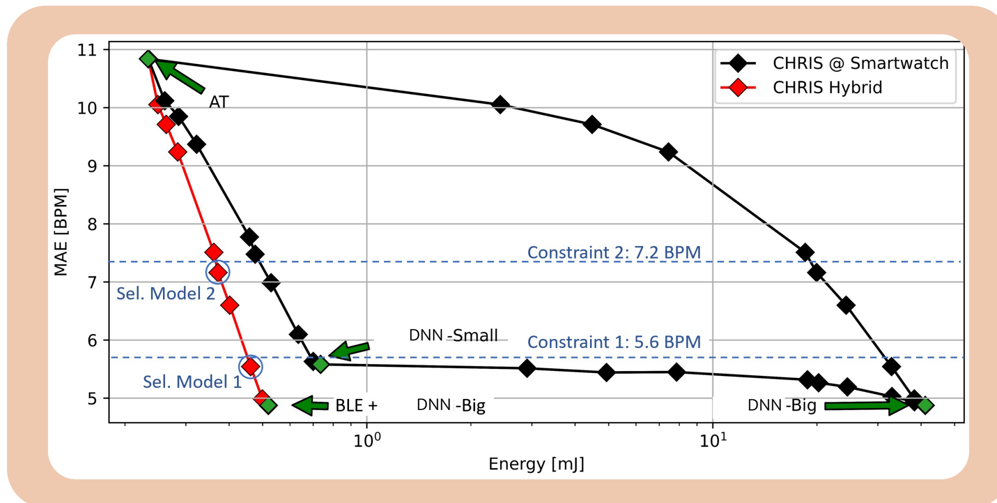

Fig. 7.4 Configurations of CHRIS in the MAE vs. Energy space.

Big) is consistently used for these offloaded tasks, yielding the lowest possible MAE. The intermediate solutions provided by CHRIS, which combines two HR tracking algorithms, are represented by red and black diamonds connecting individual models. Each diamond is associated to a different difficulty level. In this experiment, the hybrid approach that combines local execution of the AT model with remote execution of DNN-Big (indicated by the red points) proves to be Pareto-dominant compared to the other combinations tested. Therefore, when presented with a user-defined limit on MAE or energy consumption, CHRIS identifies the optimal point on this dominant front. It selects the highest point allowable under an MAE constraint or the rightmost point allowed by an energy constraint. For example, consider an MAE constraint of 5.60 BPM (matching the performance of the DNN-Small model run locally), depicted as *Constraint 1*. CHRIS would choose the operating point labeled *Sel. Model 1*. This selection corresponds to using the AT and DNN-Big models with a difficulty threshold of 8, leading to about 80% of the data windows being processed on the phone. This configuration achieves an MAE of 5.54 BPM while reducing the smartwatch's energy use by 2.03 times compared to running DNN-Small locally. If the MAE constraint is relaxed to 7.2 BPM (*Constraint 2*), CHRIS adapts by selecting the configuration *Sel. Model 2*, associated with a difficulty threshold of 6. This further lowers the average energy cost to 179 µJ per prediction. This energy level is 3.03 times lower than executing DNN-Small on the smartwatch and 1.82 times lower than streaming all input data to the phone. Lastly, if we exclude the red points from the feasible solutions (e.g., when the BLE connectivity is not available), CHRIS

by combining AT with DNN-Small or DNN-Small with DNN-Big still includes 19 Pareto points. These points span from 0.234 mJ to 41.07 mJ of energy consumption and from 4.87 BPM to 10.99 BPM of MAE.

7.2 Visual-pose Estimation on Nano-Drones

Nano-scale unmanned aerial vehicles (UAVs) are characterized by dimensions under 10 cm and weights below 40 g. Their small size enables operation in challenging scenarios, such as narrow spaces without GPS signals or close to people. Furthermore, their relatively simple sensory, mechanical, and computational components contribute to lower costs than larger drones. However, achieving full onboard autonomy without reliance on external infrastructure or computation remains challenging due to stringent hardware limitations, particularly computational power often restricted to less than 100 mW, which is significantly less (by one to two orders of magnitude) than typical mobile phone processors.

These severe resource constraints make it impractical to employ standard algorithmic pipelines commonly used in robotics, which often depend on memory-intensive and computationally demanding planning, localization, or mapping algorithms, as well as large pre-trained CNNs for perception. Consequently, deploying such methods on nano-UAVs is generally not feasible. In this setting, compact CNN architectures increasingly establish the SotA for autonomous nano-drone capabilities [167, 9]. Their suitability stems from several factors: they can be effectively trained using data from resource-limited sensors like low-power, low-resolution cameras; their computational and memory demands during inference are predictable and fixed; and system designers can tailor these requirements by carefully selecting the network architecture to match the platform's limited resources.

The proper design of a suitable, custom CNN architecture is critical for nano-scale systems. The chosen architecture directly dictates whether the model can even execute on the robot's hardware and significantly influences two key operational parameters: the accuracy of its predictions and its real-time throughput.

In this section, we exploit the lightweight PIT technique of Ch. 3, adapted to 2D CNNs as described in Ch. 5, to generate Pareto-optimal CNN architectures in the accuracy vs. model size, to optimize a vision-based human pose estimation task on nano-drones.

7.2.1 Proposed Method

Architecture Optimization

We consider three SotA alternatives as seed networks: the first is based on the shallow PULP-Frontnet architecture designed specifically for this task, while the other two are based on the MobileNetv1 [30] architecture. We consider the basic MobileNetv1 with width multipliers of 1.0 ($M^{1.0}$) and 0.25 ($M^{0.25}$).

These seeds are optimized using PIT to solve the usual optimization problem of Eq. 2.11. The employed regularization loss $\mathcal{R}$ is the differentiable estimate of the number of parameters of the network.

System Design

The robotic platform used in this work is the Bitcraze Crazyflie 2.1[1], a nano-quadrotor weighing 27 g. This drone incorporates a primary STM32 microcontroller for essential functions like state estimation and control loop execution. For our field deployment, this platform is also equipped with the commercially available AI-deck [167], a 5 g companion board. The AI-deck includes the GAP8 SoC presented in Sec. 2.3.2. Communication between the STM32 and GAP8 processors occurs over a bidirectional UART interface.

The deployment of the CNN relies on the open-source PULP-NN library [88], which provides highly optimized arithmetic kernels for the target processor. Specifically, PULP-NN leverages the eight general-purpose cores within the GAP8 SoC's cluster and utilizes ISA-specific instructions to maximize computational throughput. While the theoretical peak performance is 32 MAC/cycle (based on 4 int8 SIMD operations per cycle across each of the eight cores), PULP-NN achieves an actual peak of 15.6 MAC/cycle on square-shaped input images. This peak performance is obtained on efficient data reuse within the processor's register file, which minimizes the number of memory load and store operations per multiply-accumulate. Without such register reuse, requiring two loads for inputs and one store for the result per MAC operation, the effective peak would drop to 8 MAC/cycle. A key constraint of the PULP-NN kernels is that they are designed to operate on data residing entirely within the shared 64 kB L1 memory, restricting their direct application to relatively small network layers.

[1] https://www.bitcraze.io/products/crazyflie-2-1/

To overcome this L1 memory limitation, we utilize a second open-source tool named DORY [102]. DORY automatically generates C code based on predefined templates. This generated code effectively wraps the low-level PULP-NN kernels, handles data movement across the different memory levels of the GAP8 processor (L1, L2, and external RAM), and orchestrates the overall tensor flow. DORY employs a tiling strategy, breaking down larger network layers into smaller computational nodes whose associated data tensors can fit entirely within the L1 memory. This allows the highly optimized PULP-NN kernels to be executed directly on these smaller nodes. Specifically, DORY produces C routines that are responsible for executing the kernels on data present in L1 and managing the double-buffered transfer of tensor data from L2 memory into L1, ensuring that data is available when required for computation. Because the DMA controller operates asynchronously, these data transfers overlap kernel execution, maximizing efficiency.

Our trained models are integrated into the same closed-loop control architecture previously presented in [9]. The objective of this system is to enable the nano-drone to maintain a specific 3D position relative to the person it is observing. This involves four principal components working together: first, the deployed CNN provides an instantaneous pose estimate based on a single input image; second, a Kalman filter processes these raw estimates to produce smoother, temporally consistent pose sequences; third, a velocity controller determines the drone's target pose based on the subject's estimated current pose and calculates the necessary velocity commands to guide the drone towards that target. Finally, Crazyflie's onboard low-level control system handles motor actuation and basic flight stabilization based on these commands.

7.2.2 Experimental Results

PIT Pareto analysis

Fig. 7.5 presents the CNN architectures identified by our application of the PIT algorithm. The plot compares inference latency, measured in clock cycles, against the MAE. MAE is calculated as the sum of L1 errors across the four pose outputs (x, y, z, ϕ) between network predictions and ground truth values. A simple baseline,

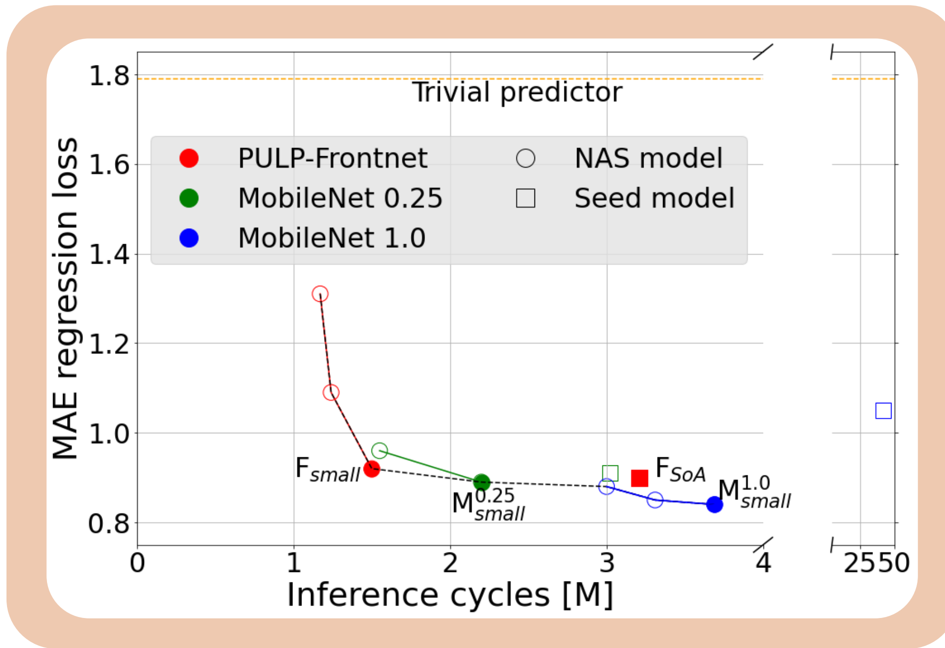

Fig. 7.5 Pareto curves of the networks extracted using PIT in the clock cycles vs. MAE space (lower is better).

the *trivial predictor* (which constantly outputs the mean value observed in the test set for each coordinate), is a lower reference point for model MAE. The PIT search yielded eight novel models starting from the three initial seed architectures. Most of these discovered architectures define the global Pareto front in the trade-off space between MAE and execution cycles. These generated models span from 1.27M to 3.69M execution cycles, achieving corresponding MAE values between 1.31 and 0.84. Specifically, models derived from PULP-Frontnet populate the leftmost segment of the Pareto curve, indicating very low latency but comparatively higher error. Architectures originating from the MobileNet 0.25× seed occupy the central region. Finally, the highest accuracy models are descendants of the MobileNet 1.0× seed. Although this initial seed architecture was too large to fit within the GAP8's L2 memory, the PIT algorithm successfully reduced its size sufficiently to produce excellent solutions, including the absolute best-performing models in terms of accuracy, which incur only a 15% latency increase compared to the baseline F_{SotA} model [9].

Four models are selected from the global Pareto front to be deployed. The selected models are: F_{SotA} which is the current SotA and corresponds to the original PULP-Frontnet; F_{small}, i.e., the fastest model, achieving an MAE similar to F_{SotA};

Table 7.4 Test set experiment results

Network	**MAE**				**R2 score**			
	x	y	z	ϕ	x	y	z	ϕ
Trivial	0.46	0.61	0.17	0.55	0.0	0.0	0.0	0.0
$\mathbf{F}_{SotA}$ **[9]**	0.19	0.18	0.09	0.44	0.80	0.65	0.55	0.26
$\mathbf{F}_{small}$	0.20	0.20	0.09	0.44	0.79	0.65	0.51	0.24
$\mathbf{M}_{small}^{0.25}$	**0.17**	0.20	0.08	0.44	**0.84**	0.63	0.62	0.22
$\mathbf{M}_{small}^{1.0}$	0.17	**0.18**	**0.07**	**0.42**	0.83	**0.67**	**0.62**	**0.28**

$M_{small}^{1.0}$, i.e., the most accurate (but slowest) architecture discovered with PIT;$M_{small}^{0.25}$, i.e., a balanced trade-off between latency and MAE.

7.2.3 In-field experimental evaluation

We evaluated the proposed models in a closed-loop, in-field experiment. A human subject followed a predefined path while the drone was tasked with maintaining a constant distance of $\Delta = 1.3\,\text{m}$ in front of the subject, aligned at eye level. The test setup matches that of [9], consisting of a 50 s trajectory divided into eight segments of increasing difficulty, including straight walking in multiple directions, curved paths, and in-place rotations. The subject was instructed to ignore the drone and synchronize each step with a metronome to ensure consistency across runs.

Table 7.5 In-field experiment results

Network	**Flight time [s]**	**Completed path [%]**	**MAE**			**Mean pose error**	
			x	y	ϕ	e_{xy} [m]	e_θ [rad]
Mocap	165	100	0.0	0.0	0.0	0.18	0.21
$\mathbf{F}_{SotA}$ **[9]**	140	85	0.33*	0.12*	0.77*	0.72*	0.78*
$\mathbf{F}_{small}$	58	35	0.81*	0.53*	0.55*	0.65*	0.42*
$\mathbf{M}_{small}^{0.25}$	**165**	**100**	**0.25**	**0.11**	**0.52**	**0.49**	**0.59**
$\mathbf{M}_{small}^{1.0}$	**165**	**100**	0.31	0.13	0.52	0.58	**0.59**

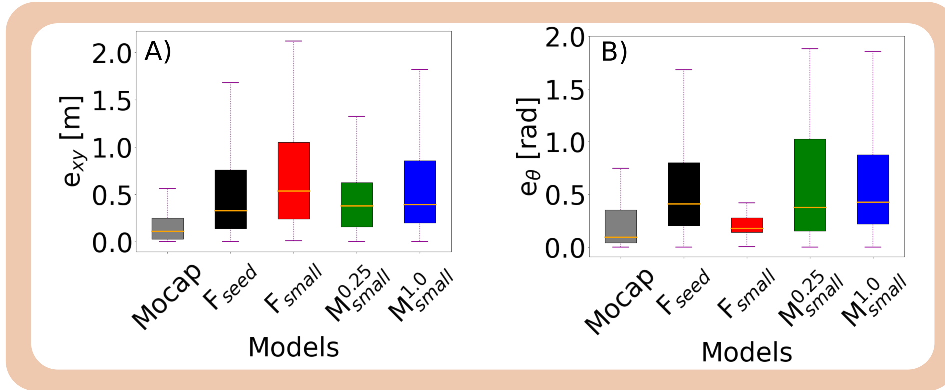

Fig. 7.6 In-field control errors distribution (lower is better). Boxplot whiskers mark the 5th and 95th percentile of data.

The experiment was conducted using both a subject and an environment not included in the training data to assess generalization. Each model was tested in four separate runs. An additional baseline run was performed using ground-truth subject pose data from a motion capture (Mocap) system for 17 flights.

Results are reported in Table 7.5. We evaluate three performance metrics: path completion, inference accuracy, and control accuracy. Path completion is measured as the cumulative flight time and the mean percentage of the trajectory completed across the four runs. A run is terminated if the subject exits the camera field of view. In this setting, both PULP-Frontnet models show limited robustness. F_{small} completes only 35% of the path on average. In contrast, both MobileNet-based models complete the full trajectory in all runs. Metrics corresponding to incomplete runs are marked with an asterisk, as they are not directly comparable.

Inference accuracy is measured independently from control. It evaluates the model's ability to estimate the subject's relative position in the camera frame, excluding the z axis due to constant target height. Accuracy decreases in this new environment. However, $M^{0.25}_{small}$ achieves the best in-field regression results, outperforming $M^{1.0}_{small}$, which had higher accuracy on the static dataset. This may be due to reduced model capacity, which can limit overfitting and improve generalization.

Finally, control accuracy quantifies how closely the drone follows the reference trajectory. We compute the position error e_{xy} in the horizontal plane and the angular error e_θ. $M^{0.25}_{small}$ performs best on both metrics. Its variance in e_{xy} is also the lowest, as shown in Fig. 7.6, indicating more stable behavior.

7.3 People-Counting on Low-Resolution Infrared Arrays

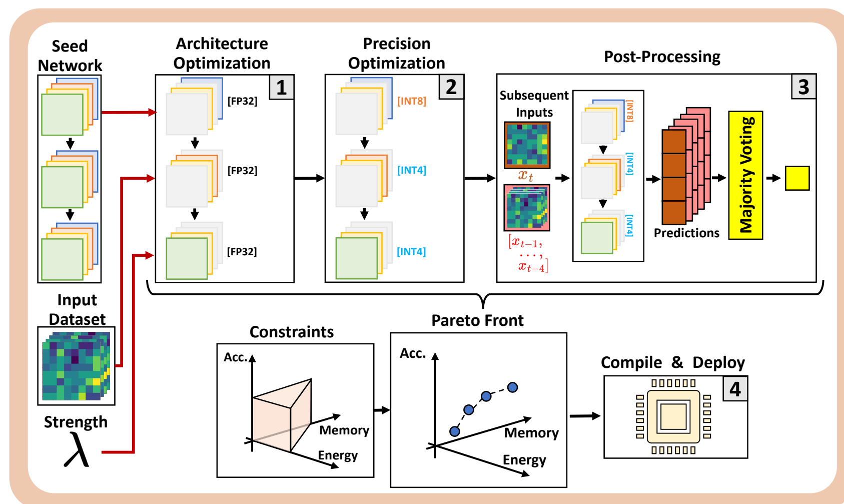

Fig. 7.7 Overview of the full-stack optimization flow.

Accurate people counting in public and private spaces is increasingly important in applications such as smart buildings and smart cities, where monitoring occupancy and flow is critical [168].

Traditional systems rely on video input processed by DNNs [169, 170]. However, these systems collect high-resolution visual data, raising significant privacy concerns in public environments [169, 170]. To address this issue, low-resolution infrared (IR) sensors have been proposed [168, 171]. These sensors capture thermal patterns instead of visual content, enabling anonymous people counting.

In addition to privacy, energy efficiency is a key design goal, particularly in deployments with limited access to the power grid. In-sensor computing reduces energy consumption by avoiding costly data transfers over wireless networks [172]. However, as extensively discussed, executing DNNs directly on sensor nodes presents challenges due to the limited computing and memory resources available at the edge.

The study in [171] explored efficient DNNs for people counting using low-resolution IR inputs from the LINAIGE [173] dataset, identifying a set of Pareto-optimal architectures. A significant limitation of this approach lies in its reliance on manual selection of architectural parameters. This manual tuning is prone to bias

and sub-optimal results, and it limits the search to coarse design spaces due to time constraints. Consequently, the search process is inefficient, yielding a low ratio of optimal models to total evaluated candidates (e.g., 0.8% in [171]).

This section addresses these issues by introducing a new automated optimization framework for IR-based people counting at the sensor level. The framework is tailored for MAUPITI, a new smart sensor platform extending the IBEX [92] RISC-V core with support for low-precision vector operations (Ref. Sec. 2.3.6). The optimization flow is outlined in Fig. 7.7, and includes the following components:

- A complexity-aware mask-based DNAS based on PIT for 2D CNNs (Ref. Ch. 3 and Ch. 5) that explores architectural hyper-parameters automatically.
- Mixed-precision quantization of weights and activations using `INT4` and `INT8` formats (Ref. Ch. 4).
- A post-processing method that exploits temporal correlation across inputs to improve accuracy with minimal overhead.

The proposed methodology produces a diverse set of Pareto-optimal models, covering nearly an order of magnitude in memory footprint.

7.3.1 Proposed Method

The objective of the flow shown in Fig. 7.7 is to generate a diverse set of DNN models for IR-based people counting, offering different trade-offs between accuracy and hardware cost. Hardware cost is measured as the number of parameters (a proxy for memory) or the number of MAC operations (a proxy for energy). The flow takes three inputs: a labeled training dataset, a seed DNN model that acts as a blueprint for generating optimized variants, and a scalar λ to balance the trade-off between accuracy and cost.

The architecture search step uses the same PIT setup for 2D CNNs described in Sec. 7.2. This allows the use of existing high-performing models as seeds. In this case, we start from the architectures manually selected in [171] and show how a finer-grained search yields improved results.

After architectural optimization, we apply quantization to reduce numerical precision from floating-point to integer formats. BN layers are folded into preceding

convolutional or linear layers to reduce operations. Each tensor T (weights or activations) is quantized using affine quantization as described in Eq. 2.2.3. We apply range-based quantization for weights and learnable quantization for activations, followed by QAT to recover lost accuracy due to BN folding and low-bit quantization.

As described in Sec. 2.3.6, the MAUPITI platforms supports `INT4` and `INT8` data formats. Mixed-precision quantization is applied at the layer level, with the same bit-width for weights and activations within each layer. Independent weight/activation or channel-wise assignments and 0-bit precision, as explored in Ch. 4, are not considered here due to hardware constraints. Indeed, to minimize HW overheads, MAUPITI only supports 4x4-bit and 8x8-bit vectorial MAC operations, i.e., the precision assignment can be different for different layers but must be the same for weights and activations of the same layer. Given the limited size of the design space (detailed in Sec. 7.3.2), we exhaustively explore all valid configurations.

The third stage of the flow is a post-processing step based on mode inference (majority voting). This exploits temporal consistency by applying the same classifier to a sequence of frames and selecting the most frequent class prediction within a sliding window. This reduces output variance and suppresses isolated misclassifications. The method has minimal overhead: unlike [171], where the model is re-executed multiple times, we reuse stored predictions in a FIFO queue, avoiding redundant computation. As a result, the energy and latency overheads are negligible, and memory requirements are equivalent to single-frame classification.

7.3.2 Experimental Results

Setup

We evaluate our optimization flow on the publicly available LINAIGE [173] dataset, designed for people counting using 8×8 IR sensor data. Despite its lower resolution compared to the MAUPITI sensor, it is the largest dataset of its kind. LINAIGE includes 25110 labeled samples across five sessions, each collected in a distinct environment. Labels indicate the number of people in view, ranging from 0 to 3. We follow the same cross-validation scheme used in [171], keeping Session 1 (the largest) in the training set for all runs and rotating Sessions 2–5 as test sets. All models are trained for 500 epochs using the Adam optimizer, a learning rate of 0.001,

batch size of 128, and cross-entropy loss. PIT and QAT are applied initially on Session 1 only. For each cross-validation fold, QAT and fine-tuning are performed on the training split (all sessions except one), and evaluation is done on the held-out session. We report performance using the average recall or Balanced Accuracy Score (BAS). Unlike [171], which runs each experiment once, we repeat each run ten times with different seeds and report mean and variance to capture statistical variability.

The PIT seed model is the largest CNN used in [171], consisting of two convolutional layers (both 3×3, stride 1, 64 channels) separated by a max-pooling layer. This is followed by two linear layers with 64 and 4 output units, respectively. BN follows each convolution. ReLU is used as activation function, except in the output layer. The goal is to show that starting from the same baseline, our automated flow can match or outperform the prior approach through finer-grained model search.

All code is implemented in Python using PyTorch.

Architecture and Precision Space Exploration

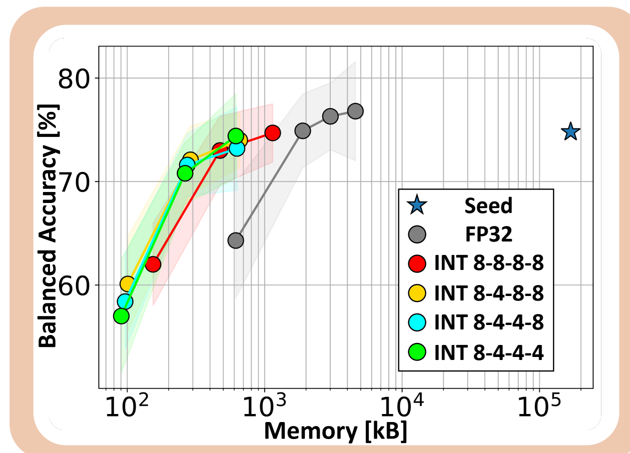

Fig. 7.8 Architecture and Precision Search Space exploration results. Different colors encode the different precisions' configurations.

The input to our flow is marked with a blue star in Fig. 7.8, shown in the BAS vs. memory plane. Starting from this point, we apply PIT using different values of the trade-off parameter λ, with the number of parameters as the cost term $\mathcal{R}$ (see Eq. 2.11). This produces the grey Pareto front. Compared to the seed, some solutions achieve up to $89\times$ reduction in memory and $26.7\times$ reduction in MACs with no drop in BAS.

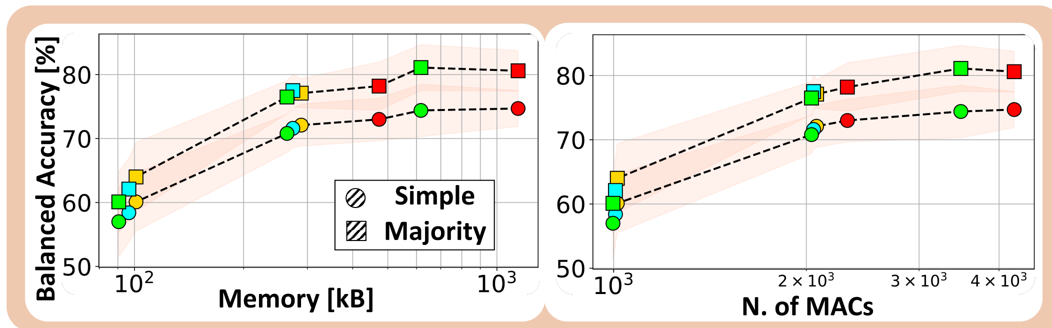

Fig. 7.9 Comparison of Pareto frontiers with and without post-processing.

The second step applies mixed-precision quantization to the floating-point models obtained with PIT. Results are shown in Fig. 7.8 as colored circles, where each color corresponds to a specific precision configuration across the four layers. Only configurations where the first layer uses INT8 are shown, as using INT4 for inputs resulted in significant accuracy loss. The “INT 8-8-4-8" variant is omitted, as it is never Pareto-optimal. Quantization improves memory by up to 2.3× and increases BAS by up to 6.5% over the lowest-memory floating-point model. Relative to the original seed, the best quantized models reduce memory and MACs by up to 147× and 234× at equal BAS.

Post-Processing Results

Fig. 7.9 reports the effect of applying the post-processing method on the best models obtained from the architecture and quantization steps. Circles represent the global Pareto front combining all optimal quantized solutions from Fig. 7.8, while squares show the corresponding results after post-processing. The two plots demonstrate that the majority-voting scheme improves the Pareto front in both BAS vs. memory and BAS vs. MACs. A sliding window of 5 predictions was used, which showed the best performance on this dataset. This method introduces a delay in response equal to half the window size (assuming correct predictions) but yields BAS gains of up to 6.7% without increasing memory or MACs.

State-of-the-Art Comparison

Fig. 7.10 compares the models obtained with our pipeline against the results reported in [171]. The left plot shows BAS versus memory, and the right plot shows BAS

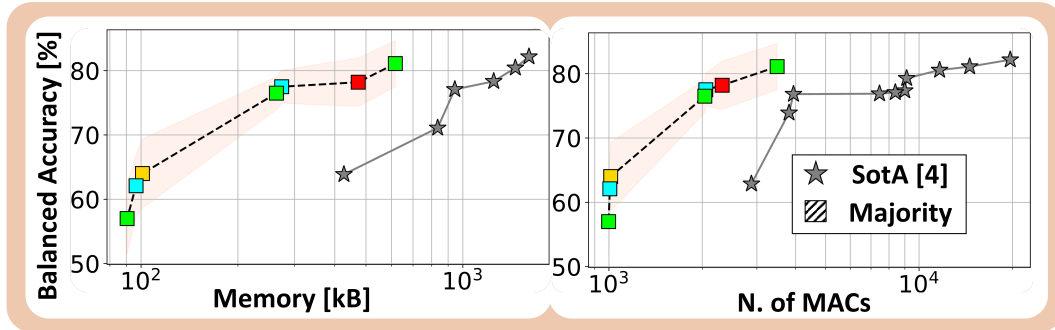

Fig. 7.10 State-of-the-Art Comparison.

versus the number of MACs. While [171] reports a slightly higher peak BAS (+0.9%), this difference falls within the standard deviation of our most accurate model. In contrast, for BAS values above 80%, our models are up to 2.4× smaller and require 3.3× fewer MACs. Compared to the most memory-efficient and lowest-MAC models in [171] (leftmost points of the grey curves), our models achieve the same BAS with 4.2× less memory and 2.9× fewer MACs.

Embedded Deployment Results

Table 7.6 Deployment results.

Model	Platform	Code [B]	Data [B]	Energy [μJ]
Top	STM32	22840	10420	9.381
	IBEX	3476	1104	6.003
	MAUPITI	4152	1104	4.927
- 5%	STM32	22970	9060	6.854
	IBEX	3384	648	5.005
	MAUPITI	4052	648	4.525
Mini	STM32	22950	8410	5.640
	IBEX	2700	416	4.342
	MAUPITI	3208	416	4.067

Table 7.6 presents deployment results for three selected models: the most accurate (*Top*), the smallest model with a BAS drop below 5% (*-5%*), and the smallest overall (*Mini*), which also has the lowest MAC count. For the architectures mentioned above, we report the code size (Code), the memory occupation (Data), and the energy consumption per inference on three different platforms. We report code size (Code), memory usage (Data), and energy per inference for each model across three

platforms. The MAUPITI system is evaluated against a vanilla IBEX core without custom instructions (Ref. Sec. 2.3.6). The code for both platforms is compiled with a custom GCC-based RISC-V toolchain using identical flags. Additionally, we compare against an STM32L4R5 MCU using 8-bit quantized models deployed with ST's X-CUBE-AI, which lacks mixed-precision support. The *Top* model shows the most significant improvement over the IBEX baseline for energy. Despite a 7% area and 2.2% power overhead, MAUPITI achieves up to 17.9% energy savings. These gains are limited by the small number of output channels in the evaluated layers, which restrict the benefits of MAUPITI's low-precision SIMD execution.

In terms of memory, MAUPITI increases code size relative to IBEX, primarily due to more complex control logic in the vector kernels. For example, the *Top* model adds 676 B. This overhead arises from the need to manage non-multiple-of-4(8) dimensions when executing SIMD instructions on irregular tensor shapes.

Compared to STM32, MAUPITI achieves up to 6.78× and 20.22× reductions in code and data size, respectively, while fitting comfortably within 16 kB memory constraints. Although STM32 executes DNNs up to 9× faster—due to its higher clock speed (120 MHz vs. 20 MHz), ISA differences, and X-CUBE-AI optimizations such as max-pooling fusion—this comes at the cost of 13.2× higher power consumption, making MAUPITI up to 1.4×–1.9× more energy-efficient.

Finally, comparing with the deployment data in [171], their smallest model (leftmost grey star in Fig. 7.10) has a 23.8× larger code size and 15.38× higher energy consumption than our equally accurate network (third square from the left in Fig. 7.10). Similarly, against their most accurate model (rightmost grey star), our *Top* model achieves a 69× code size reduction and 24.4× energy savings, with only a 0.9% BAS difference.

Chapter 8

Conclusion

This thesis presented a set of techniques to automate the optimization of DNNs' inference phase in a hardware-aware manner. The considered techniques include structured pruning, NAS, quantization, MPS, and mapping of DNNs on heterogeneous computing platforms. These techniques have been validated on various benchmarks and hardware platforms ranging from general-purpose single and multi-core MCUs to specialized accelerators.

In Ch. 3, the PIT algorithm is proposed. PIT is a lightweight gradient-based structured pruning algorithm tailored for TCNs, which can explore a large, fine-grained search space of architectures with GPU time and memory requirements similar to regular training. To our knowledge, PIT is the first tool explicitly designed for the key hyper-parameters of 1D convolutional networks, i.e., the receptive field and the dilation factor. PIT has been validated over four real-world benchmarks. It is shown that PIT can find improved versions of SotA TCNs. In particular, such solutions, when deployed on commercial edge devices, achieve a memory compression of up to 8.03$\times$ (90.8$\times$) and latency and energy reduction of up to 5.45$\times$ (19.6$\times$) without (with a reasonable) accuracy drop.

In Ch. 4, we proposed a novel optimization method for DNNs that can explore channel-wise structured pruning and MPS in the same search loop in a gradient-based and hardware-aware manner. This results in a speed-up of the optimization procedure by avoiding the time-consuming separate and sequential application of pruning and quantization techniques. Moreover, this approach avoids the restrictions dictated by the choices of the first optimization technique applied, thus allowing the

exploration of a broader search space. The proposed method achieves at iso-accuracy up to 56.17% size reduction compared to a previous SotA method. We obtain a size reduction of up to 47.50% and 69.54%, without accuracy drops, with respect to 8-bit and 2-bit fixed-precision DNNs, respectively.

In Ch. 5, we proposed DUCCIO, a new formulation of the common DNAS gradient-based optimization problem that employs multiple hardware-related constraints to find the desired network in one shot. We applied DUCCIO to two different DNAS methods, mask-based and path-based, testing on five benchmarks. We found networks that reduce the number of parameters and OPs by 55.9% and 54.6% while being iso-accurate with the baseline.

In Ch. 6, ODiMO is introduced. ODiMO is a tool that maps with fine-grain DNN execution among multiple computing units implementing different layer alternatives or with incompatible quantization formats. The mapping problem is formulated as a cost-aware differentiable optimization. This optimization problem deals with mapping parameters and standard DNN weights. ODiMO can obtain rich Pareto-fronts on different benchmarks and DNN architectures for accuracy vs. energy or latency. In particular, it reduces latency/energy by up to 8×/50.8× while incurring only limited accuracy loss compared to heuristic mappings or single-accelerator baselines.

Finally, Ch. 7 presents three real applications: PPG-based heart rate estimation on wearables, visual-pose estimation on nano-drones, and people counting on low-resolution infrared arrays. The common characteristic of these applications is that all deal with DNN inference on resource-constrained edge devices. For each task, we show how we used some of the techniques discussed in this thesis to achieve highly optimized DNN inference.

References

[1] ST Microelectronics. STM32H7.

[2] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. Gap-8: A risc-v soc for ai at the edge of the iot. In *Proc. IEEE 29th ASAP*, pages 1–4. IEEE, 2018.

[3] Gianmarco Ottavi, Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. A mixed-precision risc-v processor for extreme-edge dnn inference. In *Proc. IEEE Comput. Soc. Ann. Symp. on VLSI*, pages 512–517, 2020.

[4] Arpan Suravi Prasad, Luca Benini, and Francesco Conti. Specialization meets flexibility: A heterogeneous architecture for high-efficiency, high-flexibility ar/vr processing. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[5] Kodai Ueyoshi, Ioannis A Papistas, Pouya Houshmand, Giuseppe M Sarda, Vikram Jain, Man Shi, Qilin Zheng, Sebastian Giraldo, Peter Vrancx, Jonas Doevenspeck, et al. Diana: An end-to-end energy-efficient digital and analog hybrid neural network soc. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3. IEEE, 2022.

[6] Angelo Garofalo, Yvan Tortorella, Matteo Perotti, Luca Valente, Alessandro Nadalini, Luca Benini, Davide Rossi, and Francesco Conti. Darkside: A heterogeneous risc-v compute cluster for extreme-edge on-chip dnn inference and training. *IEEE Open Journal of the Solid-State Circuits Society*, 2:231–243, 2022.

[7] Tommaso Polonelli, Lukas Schulthess, Philipp Mayer, Michele Magno, and Luca Benini. H-watch: An open, connected platform for ai-enhanced covid19 infection symptoms monitoring and contact tracing. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[8] Matteo Risso, Chen Xie, Francesco Daghero, Alessio Burrello, Seyedmorteza Mollaei, Marco Castellano, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Hw-sw optimization of dnns for privacy-preserving people counting on low-resolution infrared arrays. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[9] Daniele Palossi et al. Fully onboard ai-powered human-drone pose estimation on ultralow-power autonomous flying nano-uavs. *IEEE IoTJ*, 9(3):1913–1929, 2022.

[10] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[11] Nahian Siddique, Sidike Paheding, Colin P Elkin, and Vijay Devabhaktuni. U-net and its variants for medical image segmentation: A review of theory and applications. *IEEE access*, 9:82031–82057, 2021.

[12] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[13] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[14] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv:1711.07128*, 2017.

[15] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[16] Luca Benfenati, Daniele Jahier Pagliari, Luca Zanatta, Yhorman Alexander Bedoya Velez, Andrea Acquaviva, Massimo Poncino, Enrico Macii, Luca Benini, and Alessio Burrello. Foundation models for structural health monitoring, 2024.

[17] Andrea Borghesi, Alessio Burrello, and Andrea Bartolini. Examon-x: a predictive maintenance framework for automatic monitoring in industrial iot systems. *IEEE Internet of Things Journal*, 10(4):2995–3005, 2021.

[18] Luca Benfenati, Sofia Belloni, Alessio Burrello, Panagiotis Kasnesis, Xiaying Wang, Luca Benini, Massimo Poncino, Enrico Macii, and Daniele Jahier Pagliari. Enhanceppg: Improving ppg-based heart rate estimation with self-supervision and augmentation, 2024.

[19] Alessio Burrello, Francesco Carlucci, Giovanni Pollo, Xiaying Wang, Massimo Poncino, Enrico Macii, Luca Benini, and Daniele Jahier Pagliari. Optimization and deployment of deep neural networks for ppg-based blood pressure estimation targeting low-power wearables. In *2024 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–5, 2024.

[20] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[21] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[22] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[23] Bharti Rana, Yashwant Singh, and Pradeep Kumar Singh. A systematic survey on internet of things: Energy efficiency and interoperability perspective. *Transactions on Emerging Telecommunications Technologies*, 32(8):e4166, 2021.

[24] Muhammed Golec, Sukhpal Singh Gill, Rami Bahsoon, and Omer Rana. Biosec: A biometric authentication framework for secure and private communication among edge devices in iot and industry 4.0. *IEEE Consumer Electronics Magazine*, 11(2):51–56, 2020.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, pages 770–778, 2016.

[26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[27] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[28] Mohamed Amine Hamdi, Francesco Daghero, Giuseppe Maria Sarda, Josse Van Delm, Arne Symons, Luca Benini, Marian Verhelst, Daniele Jahier Pagliari, and Alessio Burrello. Match: Model-aware tvm-based compilation for heterogeneous edge devices. *arXiv preprint arXiv:2410.08855*, 2024.

[29] Alessio Burrello, Matteo Risso, Noemi Tomasello, Yukai Chen, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Energy-efficient wearable-to-mobile offload of ml inference for ppg-based heart-rate estimation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[30] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[31] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1586–1595, 2018.

[32] Daniele Jahier Pagliari, Matteo Risso, Beatrice Alessandra Motetti, and Alessio Burrello. Plinio: A user-friendly library of gradient-based methods for complexity-aware dnn optimization. In *Proc. Forum Specif. Design Lang.*, pages 1–8, 2023.

[33] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, et al. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proc. IEEE/CVF CVPR*, pages 12965–12974, 2020.

[34] Matteo Risso, Alessio Burrello, Francesco Conti, Lorenzo Lamberti, Yukai Chen, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Lightweight neural architecture search for temporal convolutional networks at the edge. *IEEE Transactions on Computers*, 72(3):744–758, 2023.

[35] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[36] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[37] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proc. IEEE CVPR*, pages 2820–2828, 2019.

[38] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proc. ICML*, pages 2902–2911. PMLR, 2017.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[40] Francesco Daghero, Daniele Jahier Pagliari, Francesco Conti, Luca Benini, Massimo Poncino, and Alessio Burrello. Lightweight software kernels and hardware extensions for efficient sparse deep neural networks on microcontrollers. *arXiv preprint arXiv:2503.06183*, 2025.

[41] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: adaptive sparsity by fine-tuning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[42] Changsheng Zhao, Ting Hua, Yilin Shen, Qian Lou, and Hongxia Jin. Automatic mixed-precision quantization search of bert. *arXiv preprint arXiv:2112.14938*, 2021.

[43] Attila Reiss, Ina Indlekofer, Philip Schmidt, and Kristof Van Laerhoven. Deep ppg: large-scale heart rate estimation with convolutional neural networks. *Sensors*, 19(14):3079, 2019.

[44] Manfredo Atzori, Arjan Gijsberts, Simone Heynen, Anne-Gabrielle Mittaz Hager, Olivier Deriaz, Patrick Van Der Smagt, Claudio Castellini, Barbara Caputo, and Henning Müller. Building the ninapro database: A resource for the biorobotics community. In *Proc. 4th IEEE RAS & EMBS BioRob*, pages 1258–1265. IEEE, 2012.

[45] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv:1804.03209*, 2018.

[46] Marcello Zanghieri, Simone Benatti, Alessio Burrello, Victor Kartsch, Francesco Conti, and Luca Benini. Robust real-time embedded emg recognition framework using temporal convolutional networks on a multicore iot processor. *IEEE Trans. Biomed. Circuits Syst.*, 2019.

[47] Thorir Mar Ingolfsson, Xiaying Wang, Michael Hersche, Alessio Burrello, Lukas Cavigelli, and Luca Benini. Ecg-tcn: Wearable cardiac arrhythmia detection with a temporal convolutional network. In *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 1–4, 2021.

[48] Panagiotis Tsinganos, Bruno Cornelis, Jan Cornelis, Bart Jansen, and Athanassios Skodras. Improved gesture recognition based on semg signals and tcn. In *Proc. IEEE ICASSP*, pages 1169–1173. IEEE, 2019.

[49] Seungwoo Choi, Seokjun Seo, Beomjun Shin, Hyeongmin Byun, Martin Kersner, Beomsu Kim, Dongyoung Kim, and Sungjoo Ha. Temporal convolution for real-time keyword spotting on mobile devices. *arXiv preprint arXiv:1904.03814*, 2019.

[50] Alex Krizhevsky, Geoffrey Hinton, et al. Cifar-10, 2009.

[51] Ya Le and Xuan Yang. Tiny imagenet visual recognition challenge. *CS 231N*, 7(7):3, 2015.

[52] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[53] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset, 2019.

[54] Yuma Koizumi, Yohei Kawaguchi, Keisuke Imoto, Toshiki Nakamura, Yuki Nikaido, Ryo Tanabe, Harsh Purohit, Kaori Suefusa, Takashi Endo, Masahiro Yasuda, and Noboru Harada. Description and discussion on dcase2020 challenge task2: Unsupervised anomalous sound detection for machine condition monitoring, 2020.

[55] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[56] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[57] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[58] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[59] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proc. ICML*, pages 448–456. PMLR, 2015.

[60] Matteo Risso, Alessio Burrello, Daniele Jahier Pagliari, Simone Benatti, Enrico Macii, Luca Benini, and Massimo Poncino. Robust and energy-efficient ppg-based heart-rate monitoring. In *Proc. IEEE ISCAS*, pages 1–5. IEEE, 2021.

[61] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv:1803.01271*, 2018.

[62] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

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

[64] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[65] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[66] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journ. Mach. Learn. Res.*, 22(1), 2021.

[67] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[68] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[69] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[70] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Int. Conf. Learn. Repr.*, 2016.

[71] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[72] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *Int. Conf. Learn. Repr.*, 2017.

[73] J. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proc. IEEE Int. Conf. Comput. Vis.*, pages 5068–5076, 2017.

[74] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint, arXiv:2301.08727*, 2023.

[75] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Trans. Des. Autom. Electron. Syst.*, 27(3), mar 2022.

[76] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. Hardware-aware neural architecture search: Survey and taxonomy. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*,

pages 4322–4329. International Joint Conferences on Artificial Intelligence Organization, 8 2021. Survey Track.

[77] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, pages 6105–6114. PMLR, 2019.

[78] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.

[79] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *International Conference on Learning Representations*, 2018.

[80] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[81] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv:1805.06085*, 2018.

[82] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[83] Zhaowei Cai and Nuno Vasconcelos. Rethinking differentiable search for mixed-precision neural networks. In *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recogn.*, pages 2346–2355, 2020.

[84] Luka Macan, Alessio Burrello, Luca Benini, and Francesco Conti. Wip: Automatic dnn deployment on heterogeneous platforms: the gap9 case study. In *Proc. Int. Conf. on Compil., Arch., Synth. for Emb. Syst.*, pages 9–10, 2023.

[85] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recogn.*, 2019.

[86] Ahmed T. Elthakeb, Prannoy Pilligundla, Fatemehsadat Mireshghallah, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Releq : A reinforcement learning approach for automatic deep quantization of neural networks. *IEEE Micro*, 40(5):37–45, 2020.

[87] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[88] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A*, 378(2164):20190155, 2020.

[89] ST Microelectronics. STM32WB.

[90] Maxim Integrated. MAX30101.

[91] ST Microelectronics. LSM6DSM.

[92] lowRISC. IBEX.

[93] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv:1506.00019*, 2015.

[94] Matteo Risso, Alessio Burrello, Daniele Jahier Pagliari, Francesco Conti, Lorenzo Lamberti, Enrico Macii, Luca Benini, and Massimo Poncino. Pruning in time (pit): A lightweight network architecture optimizer for temporal convolutional networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1015–1020, 2021.

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

[96] ST Microelectornics. X-cube-ai, 2017.

[97] GreenWaves Technologies. Gap8 nntool, 2019.

[98] Yanping Chen, Yuan Hao, Thanawin Rakthanmanon, Jesin Zakaria, Bing Hu, and Eamonn Keogh. A general framework for never-ending learning from time series streams. *Data Min. Knowl. Discov.*, 29(6):1622–1664, 2015.

[99] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proc. 2016 NAACL*, pages 1480–1489, 2016.

[100] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[101] Alessio Burrello, Alberto Dequino, Daniele Jahier Pagliari, Francesco Conti, Marcello Zanghieri, Enrico Macii, Luca Benini, and Massimo Poncino. Tcn mapping optimization for ultra-low power time-series edge inference. In *Proc. IEEE/ACM ISLPED*, pages 1–6, 2021.

[102] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Trans. Comput.*, 2021.

[103] ARM. CMSIS-NN.

[104] Qian Lou, Feng Guo, Minje Kim, Lantao Liu, and Lei Jiang. Autoq: Automated kernel-wise neural network quantization. In *Int. Conf. Learn. Repr.*, 2020.

[105] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. Apq: Joint search for network architecture, pruning and quantization policy. In *Proc. IEEE/CVF Conf. Comput. Vis. Patt. Recogn.*, pages 2075–2084, 2020.

[106] Matteo Risso, Alessio Burrello, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Channel-wise mixed-precision assignment for dnn inference on constrained edge nodes. In *Proc. IEEE 13th Int. Green Sust. Comp. Conf.*, pages 1–6, 2022.

[107] Beatrice Alessandra Motetti, Matteo Risso, Alessio Burrello, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Joint pruning and channel-wise mixed-precision quantization for efficient deep neural networks. *IEEE Transactions on Computers*, 2024.

[108] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. In *Proc. 33th Int. Conf. Neural Inf. Proc. Syst.*, pages 18518–18529, 2020.

[109] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proc. 33rd Int. Conf. Mach. Learn.*, pages 2849–2858, 2016.

[110] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv:1812.00090*, 2018.

[111] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z. Pan. Mixed precision neural architecture search for energy efficient deep learning. In *Proc. IEEE/ACM Int. Conf. Comput.-Aid. Design*, pages 1–7, 2019.

[112] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[113] Ying Wang, Yadong Lu, and Tijmen Blankevoort. Differentiable joint pruning and quantization for hardware efficiency. In *Proc. Europ. Conf. Comput. Vis.*, pages 259–277, 2020.

[114] Mart Van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, and Max Welling. Bayesian bits: Unifying quantization and pruning. In *Proc. 33th Int. Conf. Neural Inf. Proc. Syst.*, volume 33, pages 5741–5752, 2020.

[115] Georg Rutishauser, Francesco Conti, and Luca Benini. Free bits: Latency optimization of mixed-precision quantized neural networks on the edge. In *Proc. IEEE 5th Int. Conf. Artif. Intell. Circ. Syst.*, pages 1–5, 2023.

[116] Krishna Teja Chitty-Venkata, Murali Emani, Venkatram Vishwanath, and Arun K. Somani. Efficient design space exploration for sparse mixed precision neural architectures. In *Proc. 31st Int. Symp. High-Perf. Parall. Distrib. Comp.*, page 265–276, 2022.

[117] Krishna Teja Chitty-Venkata, Yiming Bian, Murali Emani, Venkatram Vishwanath, and Arun K. Somani. Differentiable neural architecture, mixed precision and accelerator co-search. *IEEE Access*, 11:106670–106687, 2023.

[118] Linjie Yang and Qing Jin. Fracbits: Mixed precision quantization via fractional bit-widths. In *Proc. AAAI Conf. Artif. Intell.*, 2021.

[119] Krishna Teja Chitty-Venkata and Arun K. Somani. Neural Architecture Search Survey: A Hardware Perspective. *ACM Comput. Surv.*, 55(4):1–36, April 2023.

[120] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Trans. Design Autom. Electr. Syst.*, 27(3):1–50, May 2022.

[121] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. Mlperf tiny benchmark. In *Proc. Neural Inf. Proc. Syst. Track on Datasets and Benchmarks*, 2021.

[122] Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu. Lightnas: On lightweight and scalable neural architecture search for embedded platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.

[123] Niv Nayman, Yonathan Aflalo, Asaf Noy, and Lihi Zelnik. HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search. In *Proceedings of the 38th International Conference on Machine Learning*, pages 7979–7990. PMLR, July 2021. ISSN: 2640-3498.

[124] Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. Udc: Unified dnas for compressible tinyml models for neural processing units. *Advances in Neural Information Processing Systems*, 35:18456–18471, 2022.

[125] Matteo Risso, Alessio Burrello, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Multi-complexity-loss dnas for energy-efficient and memory-constrained deep neural networks. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED ’22, New York, NY, USA, 2022. Association for Computing Machinery.

[126] Alessio Burrello, Matteo Risso, Beatrice Alessandra Motetti, Enrico Macii, Luca Benini, and Daniele Jahier Pagliari. Enhancing neural architecture search with multiple hardware constraints for deep learning model deployment on tiny iot devices. *IEEE Transactions on Emerging Topics in Computing*, 12(3):780–794, 2023.

[127] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[128] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. Nsga-net: Neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, page 419–427, New York, NY, USA, 2019. Association for Computing Machinery.

[129] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. μNAS: Constrained Neural Architecture Search for Microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 70–79, Online United Kingdom, April 2021. ACM.

[130] Matteo Gambella, Alessandro Falcetta, and Manuel Roveri. CNAS: Constrained Neural Architecture Search. In *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2918–2923, October 2022. ISSN: 2577-1655.

[131] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. Can weight sharing outperform random architecture search? an investigation with tunas. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020.

[132] Riccardo Perego, Antonio Candelieri, Francesco Archetti, and Danilo Pau. AutoTinyML for microcontrollers: Dealing with black-box deployability. *Expert Systems with Applications*, 207:117876, November 2022.

[133] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. In *International Conference on Learning Representations*, 2018.

[134] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*, 2017.

[135] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 550–559. PMLR, 10–15 Jul 2018.

[136] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*, 2017.

[137] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. An evaluation of edge tpu accelerators for convolutional neural networks, 2021.

[138] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E. Belviranli. Axonn: Energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 1069–1074, New York, NY, USA, 2022. Association for Computing Machinery.

[139] Hongyang Jia, Hossein Valavi, Yinqi Tang, Jintao Zhang, and Naveen Verma. A programmable heterogeneous microprocessor based on bit-scalable in-memory computing. *IEEE Journal of Solid-State Circuits*, 55(9):2609–2621, 2020.

[140] Siqi Wang, Anuj Pathania, and Tulika Mitra. Neural network inference on mobile socs. *IEEE Design & Test*, 37(5):50–57, 2020.

[141] Giorgos Vasiliadis, Rafail Tsirbas, and Sotiris Ioannidis. The best of many worlds: Scheduling machine learning inference on cpu-gpu integrated architectures. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 55–64, 2022.

[142] Yuexuan Tu, Saad Sadiq, Yudong Tao, Mei-Ling Shyu, and Shu-Ching Chen. A power efficient neural network implementation on heterogeneous fpga and gpu devices. In *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, pages 193–199, 2019.

[143] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 14(1):15–18, 2022.

[144] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 615–629, New York, NY, USA, 2017. ACM.

[145] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355, February 2020.

[146] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[147] A. Di Mauro, M. Scherer, D. Rossi, and L. Benini. Kraken: A direct event/frame-based multi-sensor fusion soc for ultra-efficient visual processing in nano-uavs. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–19, Los Alamitos, CA, USA, aug 2022. IEEE Computer Society.

[148] Matteo Risso, Alessio Burrello, Giuseppe Maria Sarda, Luca Benini, Enrico Macii, Massimo Poncino, Marian Verhelst, and Daniele Jahier Pagliari. Precision-aware latency and energy balancing on multi-accelerator platforms for dnn inference. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2023.

[149] Matteo Risso, Alessio Burrello, and Daniele Jahier Pagliari. Optimizing dnn inference on multi-accelerator socs at training-time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[150] Konrad Moren and Diana Göhringer. Automatic mapping for opencl-programs on cpu/gpu heterogeneous platforms. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2018*, pages 301–314, Cham, 2018. Springer International Publishing.

[151] Ourania Spantidi, Georgios Zervakis, Sami Alsalamin, Isai Roman-Ballesteros, Jörg Henkel, Hussam Amrouch, and Iraklis Anagnostopoulos. Targeting dnn inference via efficient utilization of heterogeneous precision dnn accelerators. *IEEE Transactions on Emerging Topics in Computing*, 11(1):112–125, 2022.

[152] Ismet Dagli and Mehmet E. Belviranli. Shared memory-contention-aware concurrent dnn execution for diversely heterogeneous system-on-chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, page 243–256, New York, NY, USA, 2024. Association for Computing Machinery.

[153] Andreas Karatzas and Iraklis Anagnostopoulos. Omniboost: Boosting throughput of heterogeneous embedded devices under multi-dnn workload, 2023.

[154] Shulin Zeng, Guohao Dai, Niansong Zhang, Xinhao Yang, Haoyu Zhang, Zhenhua Zhu, Huazhong Yang, and Yu Wang. Serving multi-dnn workloads on fpgas: A coordinated architecture, scheduling, and mapping perspective. *IEEE Transactions on Computers*, 72(5):1314–1328, 2023.

[155] Mohanad Odema, Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Mohammad Abdullah Al Faruque. Magnas: A mapping-aware graph neural architecture search framework for heterogeneous mpsoc deployment. *ACM Trans. Embed. Comput. Syst.*, 22(5s), sep 2023.

[156] Halima Bouzidi, Mohanad Odema, Hamza Ouarnoughi, Smail Niar, and Mohammad Abdullah Al Faruque. Map-and-conquer: Energy-efficient mapping of dynamic neural nets onto heterogeneous mpsocs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[157] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[158] Bram-Ernst Verhoef, Nathan Laubeuf, Stefan Cosemans, Peter Debacker, Ioannis Papistas, Arindam Mallik, and Diederik Verkest. Fq-conv: Fully quantized convolution for efficient and accurate inference. *arXiv preprint arXiv:1912.09356*, 2019.

[159] D Jahier Pagliari. Deep neural network architecture search for accurate visual pose estimation aboard nano-uavs. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6065–6071. IEEE, 2023.

[160] Benjamin W Nelson, Carissa A Low, Nicholas Jacobson, Patricia Areán, John Torous, and Nicholas B Allen. Guidelines for wrist-worn consumer wearable assessment of heart rate in biobehavioral research. *NPJ Digital Medicine*, 3(1):1–9, 2020.

[161] Zhilin Zhang, Zhouyue Pi, and Benyuan Liu. Troika: A general framework for heart rate monitoring using wrist-type photoplethysmographic signals during intensive physical exercise. *IEEE Trans. Biomed. Eng.*, 62(2):522–531, 2014.

[162] Seyed Salehizadeh, Duy Dao, Jeffrey Bolkhovsky, Chae Cho, Yitzhak Mendelson, and Ki H Chon. A novel time-varying spectral filtering algorithm for reconstruction of motion artifact corrupted heart rate signals during intense physical activities using a wearable photoplethysmogram sensor. *Sensors*, 16(1):10, 2016.

[163] Alessio Burrello, Daniele Jahier Pagliari, Matteo Risso, Simone Benatti, Enrico Macii, Luca Benini, and Massimo Poncino. Q-ppg: Energy-efficient ppg-based heart rate monitoring on wearable devices. *IEEE Trans. Biomed. Circuits Syst.*, 15(6):1196–1209, 2021.

[164] D. Biswas, N. Simões-Capela, C. Van Hoof, and N. Van Helleputte. Heart rate estimation from wrist-worn photoplethysmography: A review. *IEEE Sensors Journal*, 19(16):6560–6570, 2019.

[165] Hang Sik Shin, Chungkeun Lee, and Myoungho Lee. Adaptive threshold method for the peak detection of photoplethysmographic waveform. *Computers in biology and medicine*, 39(12):1145–1152, 2009.

[166] ST Microelectornics. X-cube-ai, 2017.

[167] Daniele Palossi, Francesco Conti, and Luca Benini. An open source and open hardware deep learning-powered visual navigation engine for autonomous nano-uavs. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 604–611, 2019.

[168] Ramtin Rabiee and Johannes Karlsson. Multi-bernoulli tracking approach for occupancy monitoring of smart buildings using low-resolution infrared sensor array. *Remote Sensing*, 2021.

[169] Saleh Basalamah, Sultan Daud Khan, and Habib Ullah. Scale driven convolutional neural network model for people counting and localization in crowd scenes. *IEEE Access*, 2019.

[170] Valério Nogueira, Hugo Oliveira, José Augusto Silva, Thales Vieira, and Krerley Oliveira. Retailnet: A deep learning approach for people counting and hot spots detection in retail stores. In *SIBGRAPI*, 2019.

[171] Chen Xie, Francesco Daghero, Yukai Chen, Marco Castellano, Luca Gandolfi, Andrea Calimera, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Efficient deep learning models for privacy-preserving people counting on low-resolution infrared arrays. *IEEE IoT J.*, 2023.

[172] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 2019.

[173] Chen Xie, Francesco Daghero, and Daniele Jahier Pagliari. Linaige, 2022.