High Performance Deep Learning on Resource Constrained Platforms

Javier Fernández Marqués

Linacre College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2021

Abstract

After a decade of accelerated progress in the different areas of machine learning (ML), it has become virtually impossible to not interact with ML-powered applications and services in our everyday lives. Recommender systems, computational photography, speech translation or, getting directions are only a handful of examples of what can be done from our smartphones or other smart devices, sometimes battery-powered, present in the places where we live, work and, socialize. However, despite the known benefits of running such applications on-device (e.g. privacy, works offline, real-time potential), their reliance on millions (even billions!) of parameters and their increasingly complex execution patterns, have prevented these from leaving the Cloud, where model inference takes place. As a result, only a fraction of ML-powered applications, such as keyword spotting and next-word prediction, remain lightweight enough in terms of memory and compute footprint to run on-device.

Motivated by the urge to deploying more advanced ML-powered applications on commodity devices, the research community has primarily relied on three sets of techniques to lower their runtime costs: pruning, which results in model compression and acceleration during inference by discarding model parameters; quantization, which offers model size savings by representing parameters with fewer bits and, often translates into faster inference when using lower precision arithmetic; and, by making use of lightweight network architecture designs which better map to the target platform's hardware. However, in many cases the aforementioned techniques are insufficient to deliver the required compute, memory and energy savings to run most ML-powered applications on-device. *What further optimization avenues would enable the deployment and the efficient execution of complex and better performing models on constrained devices?*

This thesis studies this challenging problem from three different perspectives: first, by identifying a new paradigm for compression frameworks where model

parameters are generated on-the-fly, enabling inference acceleration by reducing the impact of data movement costs; second, by solving the inherent problem of numerical degradation that has prevented the adoption and deployment of fast Winograd convolutions for models making use of integer arithmetic; and third, by proposing mechanisms that enable the learning of quantized graph neural networks for a variety of applications operating on irregular grids. Each of these contributions materialize into three different frameworks, namely `unzipFPGA`, `wiNAS` and `Degree-Quant`, which are comprehensively evaluated through a wide range of experiments and ablation studies. These contributions push the state of the art on three different fronts and can be used in conjunction with other existing acceleration techniques to collectively enable the deployment of high performance deep learning applications on resource constrained devices.

Acknowledgements

During my DPhil, I have been supported in different ways by people around me. Some, I met for the first time during this 4-year journey, and others, I have known for a long time. I've been immeasurably lucky of having you all during this fun but challenging time.

I will start by thanking my wonderful advisor Nic Lane. He trusted me to become one of the first members of the OxMLSys group in late 2017, an opportunity that has profoundly changed the course of my career steering it towards the fascinating world of ML+Systems research. His constant support materialised in many ways and made my DPhil an unforgettable experience. Thank you for the very regular meetings, the access to compute infrastructure, for generously covering travel costs when attending conferences, for wisely advising me in moments of self-doubt and, for challenging me to aim high and output impactful research.

I am very fortunate of having worked alongside Milad and Catherine, two fantastic researchers that joined OxMLSys as DPhil students at the same time I did. With them, I wrote my first ML paper, attended my first conference and became a better researcher. I am also privileged of having collaborated with other members of the group: Filip, with whom I enjoyed teaching master students about ML; Edgar, with whom I investigated how to deploy ML on tiny MCUs; Vivek, who always made the group meetings interesting and fun; Shyam, thanks to whom I learned so much more about graphs and, about contemporary British culture; and Xinchi, Yan, Pedro, Titouan and Akhil, with whom I spent several months working on exciting Federated Learning projects. I look forward to collaborating with you all for many years to come!

Before our group moved to Cambridge in mid-2020, we belonged to the Cyber-Physical Systems super-group in Oxford CS. I met there a supportive and fun group of DPhil students: Shuyu, Chris, Zhihua, Linhai, Bo, Wei, Stefano, Risqi and Joe. Special thanks go to Shuyu, Chris and Zhihua: I'll never forget our

discussions about research and non-research topics, college lunches, dinners, organising Zhihua's wedding and, attending conferences.

Outside the Oxford bubble, I have collaborated with super talented researchers. In Arm, I was privileged to work alongside Paul Whatmough and Andrew Mundy as an intern. I also got to know Fernando García and Partha Maji, both made my time at Arm even more enjoyable. During my DPhil I was lucky to collaborate with two fantastic Samsung AI researchers: Sourav Bhattacharya, with whom I wrote my first publication and ventured into the domain of embedded ML; and Stelios Venieris, with whom I co-authored my last paper and learned about ML accelerators. With Daniel J. Beutel and Taner Topal, I had the opportunity to contribute to Flower, an increasingly popular open-source framework for Federated Learning. Thank you all for your advice and support.

I would like to acknowledge my *early mentors* that guided me through my first steps in research: Carlos Ortiz-de-Solórzano and Arrate Muñoz for their patient advice and support during my BSc project at CIMA; and Andrea Cavallaro for his trust and support during my MSc project at Queen Mary University. I am thankful also to Fabio Poiesi for his help during this time.

Many friends have supported me from Spain and abroad. I would like to thank specially Juan Eza, Miguel Contreras, Javier Avello, Javier Errasti and Álvaro Fuertes. Thank you for staying in touch these many years and for the great memories I'll carry with me for years to come.

My family has been a central pillar of support since I moved to the UK in 2014. The regular Skype calls, receiving packages with food and small presents and, having them visiting occasionally, made these years more enjoyable. I would like to thank my grandparents Secundino, Angelita, Jordi and Ana for their love and for making it an unforgettable experience every time I visited them. Most especially, I would like to thank my parents for teaching me the values of effort and perseverance, for encouraging me to pursue a PhD and, for their wise advice throughout the years. I thank them and my siblings Jorge, Ana and Margarita for their constant support and love. I would like to dedicate this thesis to them.

I could not end these lines without thanking XiaoYu for her love and unconditional support, without which this DPhil would not have been possible.

Contents

1 Introduction **1**

1.1 Research Questions and Contributions 3

1.2 Publications 6

2 Background **9**

2.1 Lightweight Network Designs 9

2.1.1 Pruning 10

2.1.2 Quantization 13

2.1.3 Automating Neural Architecture Search 16

2.2 On Proxy Metrics and Lower-level Optimizations 18

2.3 Fast Convolution Algorithms 20

2.4 The Data Movement Challenge 21

2.5 Graph Neural Networks 22

2.5.1 A Primer on Graph Neural Networks 23

2.5.2 Efficient Processing of Graph Neural Networks 26

2.6 Summary 28

3 On-the-fly Weights for ML Acceleration **31**

3.1 Problem Setting and Contributions 32

3.2 Background and Related Work 33

3.2.1 Compression Frameworks for CNNs 33

3.2.2 On-the-fly Compression Frameworks 35

3.2.3 FPGA-based CNN Inference Engines 37

3.3 On-the-fly Weights with OVSF Codes 40

3.3.1 Re-purposing OVSF Codes for Model Compression 40

3.3.2 On-the-fly OVSF Models: Training and Limitations 42

3.3.3 A Hardware Weights Generator using OVSF Codes 43

3.4 Experimental Setup 47

3.4.1 Datasets and Tasks 47
3.4.2 Target FPGA Platforms and Baselines 50
3.5 Experimental Results 51
3.5.1 Keyword Spotting with OVSF Models 51
3.5.2 Image Classification with OVSF Models 53
3.5.3 Reducing Data Movement with OVSF Models 55
3.6 Discussion 57
3.7 Summary 58

4 Algorithms and Architecture Search for Lightweight Inference 61
4.1 Problem Setting and Contributions 62
4.2 Background and Related Work 64
4.2.1 Winograd Convolutions in modern CNNs 64
4.2.2 Numerical Degradation in Winograd Convolutions 65
4.2.3 Hardware-aware Neural Architecture Search 66
4.3 The Winograd Algorithm for Convolutions 67
4.4 Winograd-aware Networks 69
4.4.1 Winograd-aware Convolutional Layers 70
4.4.2 Winograd-aware Neural Architecture Search 71
4.5 Experimental Setup 73
4.5.1 Vanilla Winograd-aware Networks 73
4.5.2 Evaluating Winograd-aware NAS 74
4.5.3 Winograd Convolutions on Mobile CPUs 75
4.6 Experimental Results 76
4.6.1 Winograd-aware Convolutions 76
4.6.2 Benchmarking Winograd Convolutions on Mobile CPUs 79
4.6.3 Joint micro-architecture Optimization with `wiNAS` 81
4.7 Discussion 82
4.8 Summary 84

5 Topology-aware Quantization of Graph Neural Networks 85
5.1 Problem Setting and Contributions 86
5.2 Background and Related Work 87
5.2.1 Graph Neural Networks: Overview and Challenges 88
5.2.2 Reduced precision arithmetic for Graph Neural Networks 89
5.3 Identifying the Sources of Error in GNNs 91
5.4 A topology-aware Quantization Scheme for GNNs 94

5.4.1 Topology-aware Masking of Graph Neural Networks 94
5.4.2 Percentile Tracking of Quantization Ranges 96
5.5 Experimental Setup 97
5.5.1 Datasets and Tasks 99
5.6 Experimental Evaluation 99
5.6.1 Performance with off-the-shelf Quantization Approaches 100
5.6.2 Obtaining Quantization Baselines 101
5.6.3 Performance with `Degree-Quant` 102
5.6.4 Latency and Memory Implications 103
5.7 Discussion 104
5.8 Summary 106

6 Conclusion and Future Work 107
6.1 Conclusion 107
6.2 Future Work 109

References 112

A Appendix 141
A.1 Collaboration Acknowledgments 141
A.2 Custom OVSF Code Generator 143
A.3 Comparing `unzipFPGA` to existing FPGA designs 144
A.4 Design Space Exploration for `unzipFPGA` 144
A.5 Parametrization of OVSF-based convolutional layers 147
A.6 Winograd-aware Layers for Other Architectures 148
A.7 Overhead of Learnt Winograd Transforms 149
A.8 Architectures Optimized with `wiNAS` 150
A.9 Deviation of Winograd-aware Layers 151
A.10 Additional Results for `Degree-Quant` 153

List of Figures

1.1 An overview of the main contributions presented in this thesis. First, a technique that enables the on-the-fly generation of model parameters at runtime, effectively minimizing the data movement cost [102, 324]. A hardware-based weights generator for FPGAs is proposed [333], that eases the memory boundness of single computation engines by generating model parameters on-the-fly. Second, a solution to the inherent problem of numerical degradation that has prevented the use of fast Winograd convolutions in conjunction with lightweight integer arithmetic deployments. Our Winograd-aware formulation results in $1.5\times$ speedups compare to INT8 `im2row` convolutions on Cortex-A CPUs [104]. Finally, this thesis compiles the first study [316] on the sources of degradation that uniquely arise when quantizing graph neural networks and, propose a stochastic topology-aware re-formulation of quantization-aware training suitable for graph 4

2.1 Illustration of the different types of sparsity often found on modern CNNs. Images follow a gray-scale color map where values containing zeros are shown in black. Unstructured sparsity can achieve very high compression ratios but is hard to accelerate on off-the-shelf hardware. On the other hand, structured sparsity can discard channels (shown with low opacity) to reduce the operations needed for inference, often resulting in lower latencies. Block sparsity sets tiles to zero while keeping others dense. Compared to unstructured sparsity, block sparse layers can be accelerated at lower ratios of overall sparsity on supported off-the-shelf hardware (e.g. Volta and Ampere NVIDIA GPUs) 11

2.2 Illustration of the mapping between real-valued and quantization domains for three quantization implementations given a skewed Normal distribution: (right) uniform symmetric quantization; (middle) uniform affine quantization, which allocates more bits to the left side of the distribution; and, (right) a form of non-uniform quantization which allocates more bits to regions on the real line with higher concentration of values. 14

2.3 A generic hardware-aware NAS diagram. During search, the NAS framework could be guided by introducing several elements such as: latency models measured on the target platform or amount of on-chip memory; the search space can further restricted by introducing application-level constraints such as minimum accuracy or maximum energy consumption; similarly, the search space can be expanded by considering different quantization implementations, each offering different trade offs between attainable speedup, numerical degradation and software/hardware support. 17

3.1 Overview of the main components in a typical GEMM-based CNN FPGA engine. Unrolled in memory input of dimensions $1 \times N_{in} \times H \times W$ for a convolutional layer with $K \times K$ and N_{out} output channels are loaded via Direct Memory Access (DMA) on to the respective FPGA buffers. This process is done in a $\langle T_R, T_P, T_C \rangle$ tiled fashion, generating a single $T_R \times T_C$ output tile after accumulating over $\left\lceil \frac{P}{T_P} \right\rceil$ matrix multiplications. 38
3.2 Fully constructed binary tree with OVSF codes of lengths $L \in 2, 4, 8$. Each code can be individually addressed as $C_{L,K}$ where L is referred to as *spreading factor* in the telecoms jargon and K the *code index*. For example, $C_{2,2}$ is equal to the second row in H_2 in Equation (3.3). 41
3.3 Constructing filters of a CNN layer using OVSF codes. A filter of shape $N_{in} \times K \times K$ is obtained by performing a linear combination of $\hat{L} = \lfloor \rho \cdot L \rceil$ of length $L = N_{in} \times K \times K$, with $\rho \in [0, 1]$. Then the $N \times 1$ vector is reshaped to the target filter's shape. If the convolutional layer has N_{out} output channels, this process is repeated that many times concatenating the results. 42
3.4 Overview of `unzipFPGA`'s architecture. In this example, the weights tile is split into three sub-tiles that are generated sequentially by `CNN-WGen`. The OVSF Generator uses a FIFO buffer to store OSVF codes and, logic to serve and re-insert them efficiently. The Alpha Buffer stores the $\{\alpha\}_{k=1}^{L}$ coefficients learnt during training. The combination of OVSF codes is performed in the M-wide array of multipliers and adders. Then, the corresponding sub-tile in the $T_P \times T_C$ tile is updated. The control unit (CU) resets the state of the accumulators between sub-tiles updates. . . 44
3.5 (Above) a typical Keyword Spotting system where MFCC features are first extracted from a stream of audio and are then passed to a neural network based classifier. (Below) a diagram showing the main components in `BinaryCmd`. The input convolutinal layer uses 8×4 filters while the next two on-the-fly layers use 4×4 filters. These can be perfectly constructed with OVSF codes since they are a power-of-two. The number of output channels (N) is shown in Table 3.1. 49
3.6 Results comparison against architectures in [414] for the category of *small* micro-controllers. DS-CNN has never been tuned below 38.6kB and 5.4M OPs. All other configurations result in larger and computationally more expensive models. 52
3.7 Number of OPs required for each filter generation method for different filter dimensions. 52
3.8 Execution time (seconds) required for each filter generation method for different filter dimensions on Arm Cortex-M7. 52
3.9 Speedup over optimized ResNet-18 (a-b) and ResNet-34 (c-d) baselines when varying the available off-chip memory bandwidth. On-the-fly models offer much faster inference stages in the context of restricted bandwidth. This gap compared to Tay82, which does structured pruning, becomes even larger on the Z7045, a lower-end FPGA platform. In light of these results, the on-the-fly formulation presented in this chapter becomes a good candidate for supporting multi-tenant FPGA platforms. 57

4.1 Standard convolutions operate on a tile (blue) defined by the filter size, generating a single output. Winograd convolutions operate on larger tiles without modifying the filter dimensions. A 3×3 Winograd convolution operating on a 4×4 tile (red) generates a 2×2 output. This is often expressed as $F(2 \times 2, 3 \times 3)$. 62

4.2 The `wiNAS` framework brings all the contributions presented in this chapter together. It leverages Winograd-aware layers that capture the numerical errors of Winograd convolutions during training. It uses a real latency model to guide the neural architecture search (NAS) stage which, does not change the model architecture, but jointly optimizes the choice of convolution algorithm for each 3×3 layer balancing inference speedup and numerical degradation. `wiNAS` automates this process based on the target platform and task at hand. On the right, the resulting ResNet-18 macro-architecture is shown. These diagrams are better shown in Figure A.2 in the Appendix, with `wiNAS` $_Q$ and WA$_{F4}$ being described in Section 4.5.1. 63
4.3 When transforming the filters to the Winograd domain, $GgG^{\top}$, these are made to match the input tile size. Larger tiles sizes offer greater parallelization potential at the cost of increased run-time memory usage. Smaller 3×3 filters, ubiquitous in modern CNNs, disproportionately suffer more from increasing tile sizes than 5×5 and larger filters. 69
4.4 Replacing the convolutions in pre-trained ResNet-18 models on CIFAR-10. Winograd works well in FP32, but accuracy drops drastically with quantization for larger tiles. Quantized configurations performed a *warmup* of all the moving averages in Eq.4.1. Without this relaxation, requiring a Winograd-aware layers as in fig. 4.5, $F2$ would be unusable. 69
4.5 Forward pass of Winograd-aware layers. Transformation matrices G, $B^{\top}$ and $A^{\top}$ are constructed via Cook-Toom. If these are included in the set of model parameters, they would be updated with every batch via back-progation (this is represented with the coloured arrows going back to matrices G, $B^{\top}$ and $A^{\top}$, carrying the gradients to update the values of each transform). In its default configuration, each intermediate output throughout the pipeline quantized to the same level as the input and weights, this is represented by Q_x. 70
4.6 With `wiNAS`, each 3×3 convolution in a given architecture (on the left, the first 2 residual layers of a ResNet18 network are shown) is implemented with either `im2row` or with Winograd convolutions varying tile size. While the former is lossless and faster than direct convolution, Winograd offers lower latencies but introduce numerical instability that could ultimately impact in accuracy. The first convolutional layer of the macro-architecture is fixed to use standard convolutions since, due to the large input dimensions, Winograd convolutions do not offer speedups (as shown in Figure 4.12). `wiNAS` does not change the network's macro-architecture, it optimizes the choice of convolution algorithm to balance accuracy, latency reduction and, numerical error. 72
4.7 Performance of a Winograd-aware ResNet-18 at different bitwidths and trained with different Winograd configurations. We show how Winograd-aware layers scale with network's width. In quantized settings, models that learn the Winograd transforms (*-flex* configurations), strictly outperforms those models that keep them fixed with the values obtained via Cook-Toom. 76

4.8 Distributions of weights and outputs of the 3×3 convolutional layers in the fourth residual block of a $F4$-ResNet18 network with width multiplier set to 0.25. Distributions for *static* winograd-aware layers are shown above for different bitwidths, below for *flex* layers. In black, the contour for the distributions from standard 8-bit convolutional layers. Distributions from *static* layers become wider as bitwidth is decreased for both weights and outputs. The latter is noticeable more distorted due to the errors introduced during the Winograd convolution with default transforms. This is not the case when these are learnt. The patterns shown in this figure appear across all residual blocks in the network. 77
4.9 Clustering of inputs to each layer in a CNN for MNIST (10 classes) comprised of three $F(4\times4,3\times3)$ layers followed by a linear layer. Both *flex* (below) and *static* (above) models were trained for 10 epochs and at 8-bits. The former reached 99.3% accuracy while the latter just 82.1%. After each *static* layer, the output tensor has degraded, resulting in overlapped clusters. This is not the case on *flex* layers. Once training is completed, we passed the test set (10K images) through the network and used a manifold approximation clustering mechanism [245] to project the input tensor at each layer. Each dot is a projection of a single input, there are 10K dots in each layer's projection. 78
4.10 Performance of INT8 LeNet on MNIST using standard convolutions (`im2row`) or Winograd-aware layers. Letting the transformations to evolve during training (-*flex*) always results in better models. F4 and F6 configurations (not shown) reach an accuracy of 73% and 51%, respectively. All configurations reach 99.25% $\pm$ 0.1% in full precision. 79
4.11 Transforming a standard model (trained with default convolutions) to its Winograd-aware counterpart can be done in very few epochs of retraining. We found this technique works best if the Winograd transformations are learnt during retraining (-*flex*), otherwise adaptation becomes much more challenging. 79
4.12 Latencies (in milliseconds) of convolving increasingly larger input tensors in the width/height dimensions (y axis) and in depth (x axis). We compare the time needed for `im2row` and each of the Winograd configurations with 32-bit arithmetic on a Cortex-A73. We show that (1) `im2row` is the consistently the optimal algorithm for the input layer to a network, (2) the choice between $F2$, $F4$ and $F6$ should be done based on the output's width/height and, (3) this choice should not generally be altered based on $inCh \rightarrow outCh$. 79
4.13 Latencies (normalized *w.r.t* `im2row`) to execute different layers of the ResNet-18 network measured in A73 (above) and A53 (below). For Winograds, solid colour bar regions represent the element-wise Hadamard GEMM stage, below and above it are respectively the input and output transformation costs. Measured for FP32 and with standard Winograd transforms. 80

5.1 Despite GNN model sizes rarely exceeding 1MB, the OPs needed for inference grows at least linearly with the size of the dataset and node features. GNNs with models sizes $100\times$ smaller than popular CNNs require many more OPs to process large graphs. 86
5.2 While CNNs operate on regular grids, GNNs operate on graphs with varying topology. A node's neighborhood size and ordering varies for GNNs. Both architectures use weight sharing. 86

5.3 Analysis of values collected immediately after aggregation (top two plots) and after generating node embeddings (bottom row) at the final layer of FP32 GNNs trained on Cora's validation set. The data shown has been averaged over 100 runs. For each architecture, we show the mean and variance of the aggregated messages, $\mathbf{O} \in \mathbb{R}^{|V| \times F'}$, at each node and represent it as a point in the plot. We follow the same approach to represent the node embeddings, $\mathbf{Y} \in \mathbb{R}^{|V| \times F'}$. **Top row:** As degree grows, so does the mean and variance of channel values after aggregation except in GAT. This suggest that quantization of the aggregation stage would have a more damaging effect on GIN, less on GCN and, negligible in GAT models. **Bottom row:** The features' distributions remain within an almost constant range of values independently of the node's degree. The dashed line is the result of performing a Ridge regression on the results for each architecture. 93
5.4 High-level view of the stochastic element of `Degree-Quant`. Protected (high degree) nodes, in blue, operate at full precision more often, while unprotected nodes (red) operate at reduced precision. After each of the three stages (update stage not show in the diagram) in the message-passing pipeline, the DQ mask is applied to ensure that these can be accelerated using integer arithmetic. High degree nodes contribute most to poor gradient estimates, hence they are stochastically protected from quantization more often. After training, all stages are quantized to the same bitwidth, including high degree nodes. Figure A.7 shows an equivalent diagram for QAT. 95
5.5 Maximum observable values with absolute `min`/`max` and percentile ranges, applied to INT8 GCN training on Cora. We observe that the percentile max is half that of the absolute method, *doubling* resolution for the majority of values. 102
5.6 Degradation of INT8 GCN on Cora as individual elements are converted to INT4 *without* *`Degree-Quant`*. No single stage has a large impact in final accuracy when at INT4. 104
5.7 Degradation of INT8 GIN on Cora as individual elements are converted to INT4 *without* *`Degree-Quant`*. Quantizing the `update` stage results in severe degradation. 104

A.1 Microarchitecture of the `CNN-WGen` module with a detailed representation of the OVSF code generator. In Section 3.3.3 all the elements in `CNN-WGen` are described. 143
A.2 Resulting architectures after optimizing a ResNet-18 macro-architecture using wiNAS. For wiNAS$_{\text{WA}}$ and CIFAR-100, the architecture resulted is shown on the left. With wiNAS$_{\text{WA-Q}}$, that introduces quantization in the search space, the optimization resulted in different architectures for CIFAR-10 (middle) and CIFAR-100 (right), evidencing the difference in complexity of the latter. 151

A.3 Deviation in the output of 3×3 Winograd-aware convolutional layers w.r.t standard convolution when using *static* transforms and when using *flex* transforms. We show this difference (measured as MSE) for Winograd-aware layers at INT8, INT16 and FP32. Larger plots of the left show the validation and loss accuracies of each of those layer configurations. As expected, *static* layers perform worse at lower bitwidths. The smaller plots show the MSE difference measured at the output of each Winograd-aware layer compared to that that would had been obtained using the same weights and input but following a standard convolution. The y-axis is in logarithm scale and x-axis in all plots represent the epoch number. The difference is marginal with *static* layers at FP32, as expected. This grows as bitwidth is reduced (due to the numerical errors in Winograds). For *flex* layers, even at FP32 the difference w.r.t normal convolutions is very large and, grows for INT8. 152
A.4 Analysis of how INT8 GAT performance degrades on Cora as individual elements are reduced to 4-bit precision *without DQ*. For GAT the message elements are crucial to classification performance. Similarly to GIN layers, quantizing the `update` stage results in severe degradation, although not as much as in the former. 153
A.5 Validation loss curves for GIN models evaluated on REDDIT-BINARY. Results averaged across 10-fold cross-validation. We show four DQ-INT8 experiments each with a different values for ($p_{\min}$,$p_{\max}$) and our FP32 baseline. 154
A.6 Diagrams representing how the output graph-summarization stages for graph-level tasks (e.g. graph classification, graph regression) are implemented when making use of DQ (left) and nQAT (right). GNNs making use of DQ during the node-aggregation stages (see fig. 5.4), do not use the stochastic element of DQ in the output MLP layers but still make use of percentiles. For models making use of nQAT, the final MLP still makes use of stochastic quantization of weights. 154
A.7 Diagram representing how nQAT (vanialla QAT with quantization noise for the layer weights as in [309]) is implemented for GNNs. The diagram illustrates this for a GCN layer. The stochastic stage only takes place when quantizing the weights, the remaining of the quantization modules happen following a standard QAT strategy. A QAT diagram would be similar to this one but fully quantizing the weights. . . . 155

List of Tables

3.1 Parameters for each configuration evaluated for `BinaryCmd` and associated memory and compute footprints. Parameters are given in triplets since there are three on-the-fly convolutional layers (see Figure 3.5). For KWS evaluation, we focus on the accuracy retention when training extremely small models and exclude the compute costs of generating the weights. The number of operations (OPs) for inference do not account for those costs. 49
3.2 FPGA platforms used for evaluation. The Zynq 7045 is a mid-tier FPGA while ZU7EV can be considered to be a tier higher, both are from Xilinx. On-chip memory are commonly arranged in small blocks (e.g. 36Kb in the case of the Z7045), which can be interconnected or stacked in various ways, hence the term BRAM, which stands for "Block RAM". 50
3.3 Comparison of three `BinaryCmd` configurations against DS-CNN, the current state of the art for KWS applications, and other baselines presented in [414] for MCUs limited to a maximum of 80kB of memory and 6M OPs. Values shown are for networks with 8-bit quantization. 51
3.4 Analyzing the impact on accuracy for each basis selection strategy (relevant when $\rho<1$) and method to obtain 3×3 filters from 4×4 OVSF filters. Models trained on CIFAR-10 on the commonly used ResNet-18/34 implementations for this dataset. We also test on the much smaller variation of such architectures (†) proposed by He et al. [146]. Performing an iterative drop of basis, as opposed as taking the first $\lfloor\rho \cdot L\rceil$, consistently results in better models. As model size is reduced, taking a 3×3 crop from a 4×4 filter performed better than using an average pooling stage. . 54
3.5 Accuracy and number of parameters for ResNet-18 models on ImageNet following different compression schemes. Performance measured on the Zynq 7045 platform at different memory bandwidths. 55
3.6 Accuracy and number of parameters for ResNet-34 models on ImageNet following different compression schemes. Performance measured on the Zynq 7045 platform at different memory bandwidths. 56
3.7 Comparing with faithful baseline on SqueezeNet on ImageNet. Performance measured on the UltraScale+ ZU7EV platform at different memory bandwidths. 56

4.1 Key hardware specifications for the high-performance Cortex-A73 and the high-efficiency Cortex-A53 cores found on a HiKey 960 development board. Both are mobile CPUs widely available in off-the-shelf hardware. 75

4.2 Performance in terms of accuracy and latency (ms) of ResNet-18 when convolutions are implemented with different algorithms and for different quantization levels. We show that Winograd-aware ($WA_{F2/4}$) layers combine the speedups of Winograd convolutions with those of INT8 arithmetic, with little to no accuracy loss in some cases. This is not possible with existing Winograd ($W_{F2/4}$) formulations. Latency is measured on Arm Cortex-A73 and A53 cores. For the last two rows, `wiNAS` found different optimizations for each dataset. We show latencies for CIFAR-10 on the left and CIFAR-100 on the right. Speedups are shown against `im2row` in FP32. (*) With default Winograd transforms. (†) Includes worst case latency increase due to be using learned transform, which are often dense. 82

5.1 Configuration of architectures evaluated for citation networks (Cit), MNIST (M), CIFAR-10 (C), ZINC (Z) and REDDIT-BINARY (R). We relied on Adam optimizer for all experiments. For all batched experiments, we used 128 batch-sizes. All GAT models used 8 attention heads. All GIN architectures used 2-layer MLPs, except those for citation networks which used a single linear layer. 97
5.2 Number of parameters for each of the evaluated architectures 98
5.3 Statistics for a few popular datasets found in the literature. Acronyms for tasks: NC, stands for Node Classification; GC, stands for Graph Classification; GR, stands for Graph Regression; LP, stands for Link Prediction. 98
5.4 Impact on performance of four typical quantization implementations for INT8 and INT4. The configuration that resulted in best performing models for each dataset-model pair is bolded. Hyperparameters for each experiment were fine-tuned independently. As expected, adding clipping does not change performance with min/-max but does with momentum. 100
5.5 This table is divided into three sets of rows with FP32 baselines at the top. We provide two baselines for INT8 and INT4: standard QAT and stochastic QAT (nQAT). Both are informed by the analysis in 5.6.1, with nQAT achieving better performance in some cases. Models trained with `Degree-Quant` (DQ) are always comparable to baselines, and usually substantially better, especially for INT4. 101
5.6 Results for DQ-INT8 GIN models perform nearly as well as at FP32. For INT4, DQ offers a significant increase in accuracy of over 23% compared to the stochastic QAT baseline. 102
5.7 INT8 latency results run on a 22 core 2.1GHz Intel Xeon Gold 6152 and, on a GTX 1080Ti GPU. Quantization provides large speedups on a variety of graphs for CPU and non-negligible speedups with unoptimized INT8 GPU kernels. 103
5.8 Ablation study against the two elements of `Degree-Quant` (DQ). The first two rows of results are obtained with only the stochastic element of DQ enabled for INT8 and INT4. Percentile-based quantization ranges are disabled in these experiments. The relative improvement is compared against vanilla QAT implemenation. DQ masking alone is often sufficient to achieve excellent results, specially in the case of GIN, but the addition of percentile-based range tracking can be beneficial to increase stability. 104

A.1 Comparison with existing FPGA work on ResNet18 (4.03 GOps), ResNet34 (7.40 GOps) and SqueezeNet (0.78 GOps).(*) using OVSF50, (†) 18×18, 19×18 and 25×18 DSP configurations. 145

A.2 Comparison with existing FPGA work on ResNet50 (8.41 GOps). Ar10 and St10 stand for Arria 10 and Stratix 10, respectively. The Adaptive Logic Modules (ALMs) are Intel's implementation of LUTs. (*) using OVSF50, (†) 18×18, 19×18 and 25×18 DSP configurations. 145
A.3 Value of OVSF ratio ρ assigned to each pair of 3×3 convolutional layer in each Residual Block or BottleNeck Block in ReNets. For SqueezeNet's Fire module, which contains a single 3×3 convolutional layer, we make use of the value ρ under the *conv1* columns. 148
A.4 Comparison between standard convolutions (`im2row`) and Winograd-aware layers for SqueezeNet. With INT8 quantization and using the default transformation matrices (*static*), larger tile sizes (i.e. F4) introduce substantial numerical error and result in a sever accuracy drop. This drop in accuracy is significantly reduced if the transformations are learnt (*flex*). 148
A.5 Comparison between standard convolutions (`im2row`) and Winograd-aware layers for ResNeXt_20(8×16). With INT8 quantization and using the default transformation matrices (*static*), larger tile sizes (F4) introduce substantial numerical error and result in a sever accuracy drop. This drop in accuracy is significantly reduced if the transformation matrices are learnt (*flex*). 149
A.6 Final test accuracies for FP32 and DQ-INT8 models whose validation loss curves are shown in Figure A.5 154

Chapter 1

Introduction

After a decade of accelerated progress in machine learning (ML), it is no longer a debate that ML systems have surpassed average human-level performance in concrete tasks such as image classification [145], speech recognition [380, 20] or, when playing some board and arcade games [304, 23]. These tasks in particular, have been object of sustained research and, inevitably, became the Petri dish in which the research community at large cultured novel model architectures, training techniques for better model generalization, software and hardware level optimizations and, datasets that more closely resemble real world scenarios. Moreover, image classification and speech recognition are two of the core tasks in the MLPerf benchmark [284], the industry's gold standard benchmark to fairly compare ML systems holistically and, often the single point of reference when comparing the performance of radically different commercially available ML platforms such as NVIDIA's GPUs, Google's TPUs, Graphcore's IPUs or, Habana Labs' accelerators, among others. However, the landscape of applications is much more diverse than those usually considered when referring to the term ML or *artificial intelligence* (AI). In addition to the aforementioned, well studied tasks, a plethora of other applications benefited from the ML momentum of recent years. These transitioned from using handcrafted features and sometimes overly complex, task-dependent algorithmic paradigms into a unified, data-driven formulation that allowed for easier reproducibility, comparisons and optimizations, fuelling further advancements. Such tasks include: autonomous driving [168, 50, 49], physics simulation [357, 361, 273], recommender systems [326, 419], scene reasoning [403, 388, 409], image super-resolution [321, 213, 413, 415], protein folding [296, 24, 185], chip design automation [250, 1], co-operative games [336, 214], question-answering systems [204, 416, 394, 79], automated speech recognition [271, 253, 255, 64], federated learning [246, 221, 163, 278, 110] and, many more.

This proliferation of ML-powered applications is not only, as it is commonly acknowledged, the result of a more democratized access to compute resources (e.g. GPUs) and, the availability of larger and often curated datasets [209, 199, 75, 227, 359, 68, 18, 269, 342, 279]. These efforts have been largely supported as well by a rich ecosystem of frameworks that enabled fast prototyping [272, 2, 56, 34], scheduling and parallelization of experiments [260, 178, 223], analysis and visualization of results [172, 360] and, deployment on general purpose and specialized hardware [57, 126, 287]. Additionally, the much wider range of available hardware (e.g. server-class GPUs and CPUs, their mobile counterparts, microcontrollers, FPGAs, ASICs) not only supports training but also inspires the development of novel applications. For example: battery-free devices that can harvest energy from their environment and perform basic sensing [306, 327] would make ubiquitous ML systems possible in otherwise unpractical settings (e.g. crop fields, forests); or LiDAR sensors that are now being embedded on some smartphones, will drive the development of efficient on-device point cloud processing techniques that are yet to be discovered. Such an increasingly rich ecosystem of hardware, sensors and, reduced costs for custom solutions, collectively fuel the desire for newer ML applications that better embodied in our society. However, such developments must be supported by an equivalent, if not superior, effort to design optimization strategies that would enable them to be deployed and run efficiently.

Despite the evident differences between model architectures, type of data consumed and, hardware existing ML applications run on, they are all constrained in one way or another. Specifically, it is either the latency requirements, memory footprint, energy consumption or, several of these factors the ones that have restricted wider deployment of ML-powered applications and services on commodity devices such as smartphones, wearables, smart appliances and, battery powered devices such as drones and small robots [206]. When considering training on-device, as is the case for example in Federated Learning settings, these limitations get amplified. As a result, the vast majority of ML-powered services either live in or off-load the majority of the compute to the Cloud, where powerful server-class hardware is available. Today, very few applications that run on-device, namely key-word spotting and next-word prediction, have been universally adopted.

How has the research community addressed the challenges of reducing the compute, memory and energy requirements of ML workloads that is limiting their usage more widely? With a concrete example: What set of techniques have been the main drivers that made it possible to transition from models such as 2014's VGG [305], with its 138M parameters and 15.6B FLOPS, into EfficientNet-B0 [319] five years later, that can reach 4% higher Top-1 accuracy on ImageNet while relying on $26\times$ fewer parameters and $40\times$ fewer

FLOPS? Model architecture unequivocally plays a direct role in addressing the complexity, both in terms of compute and memory, of running ML workloads. As a result, it is often the first dimension to consider when seeking high-level optimizations. This trend began with residual connections in ResNets [146], later with depth-wise convolutions popularized by MobileNets [165, 293], other hand-designed architectures [411, 173, 108, 170] and, orthogonal approaches that focused on model pruning techniques [140, 256, 143, 400, 257, 353] pursuing the removal of redundant elements of already existing architectures. However, the manual design of model architectures was overhauled by automated frameworks, collectively categorized under the neural architecture search (NAS) label [318, 47, 230, 42, 225, 164, 319]. These approaches do a better job at exploring the vast design space while considering, not only the set of candidate layers (e.g. different types of convolutions, activations, pooling layers, etc), but also their memory, compute and energy footprints. As a parallel phenomenon, another line of research focused instead on reducing the compute and memory budget required to run ML applications without changing the model architecture. This second dimension, considered the use of low precision arithmetic to accelerate model inference [175, 347, 402, 274, 71, 11, 127] while preserving the accuracy of the original, full precision model. Their adoption not only translates into faster inference stages and reduced model sizes but also results in lower power consumption and chip area requirements [162, 365]. These two orthogonal dimensions, spanning neural architecture designs and low-precision optimizations, have by far dominated the efforts of bringing applications relying on computer vision and language models closer to production-ready levels.

1.1 Research Questions and Contributions

The work presented in this thesis contributed to the pursuit of more lightweight inference stages for various types of ML workloads. But, unlike the majority of other research concentrated in the domains of lightweight architectural designs, model pruning or reduced precision arithmetic for CNNs and language models, this thesis explores orthogonal avenues while still sharing the same goal of enabling the deployment of high-performing ML-powered applications on resource constrained devices. The work here presented concentrates on: reducing data movement through lightweight on-the-fly generation of model parameters; algorithms for convolutions for lower-level compute graph acceleration; and, a topology-aware formulation for quantization aware training on graphs. Each of these three contributions address a different research challenge known by the ML and systems community to varying degrees but that remained unsolved, at least partially. Each of the

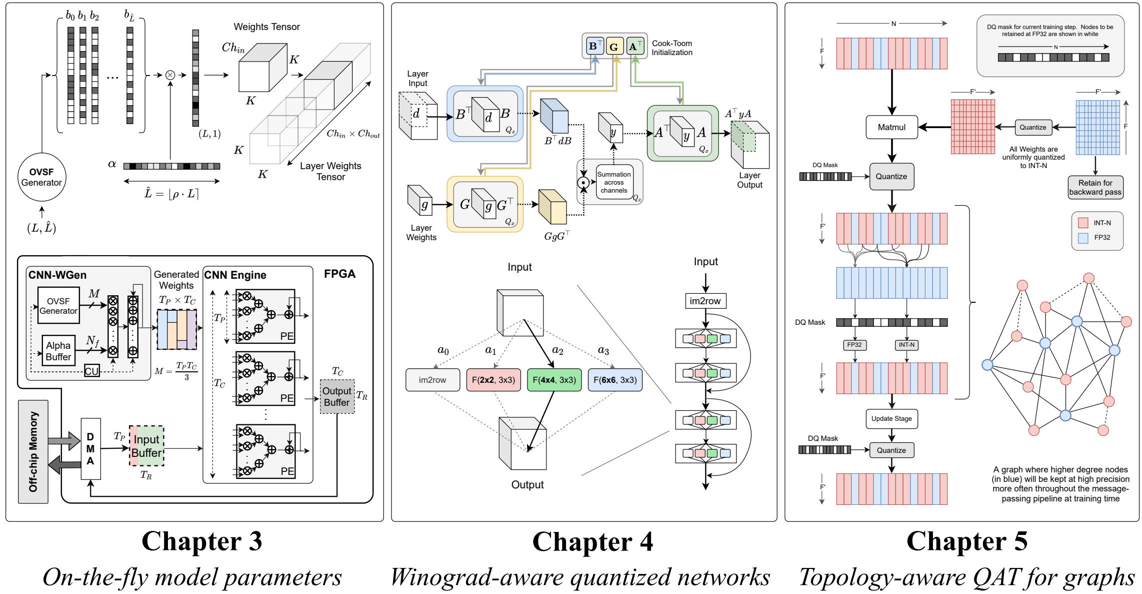


Figure 1.1: An overview of the main contributions presented in this thesis. First, a technique that enables the on-the-fly generation of model parameters at runtime, effectively minimizing the data movement cost [102, 324]. A hardware-based weights generator for FPGAs is proposed [333], that eases the memory boundness of single computation engines by generating model parameters on-the-fly. Second, a solution to the inherent problem of numerical degradation that has prevented the use of fast Winograd convolutions in conjunction with lightweight integer arithmetic deployments. Our Winograd-aware formulation results in $1.5\times$ speedups compare to INT8 `im2row` convolutions on Cortex-A CPUs [104]. Finally, this thesis compiles the first study [316] on the sources of degradation that uniquely arise when quantizing graph neural networks and, propose a stochastic topology-aware re-formulation of quantization-aware training suitable for graph

published works within this thesis provided an original solution to those problems. The three research questions that this thesis seeks to answer are:

Q1. With the wide adoption of lightweight model architectures and quantization, the once flourishing community developing compression frameworks fade out rapidly. This is because while the former translate into compact model representations and direct inference acceleration, the latter only achieve compression. *Can we design a compression framework that upon deployment decompresses the model parameters on-the-fly, effectively accelerating inference, not by doing less compute, but by reducing data movement?*

In Chapter 3, this thesis proposes a new formulation for compression frameworks which not only fixate on compression ratio but also account for the necessary decompression stage upon deployment and, include such compute and memory overheads at design time. We then define what *on-the-fly* compression methods are and, propose a lightweight technique that allows for an efficient on-the-fly generation of model parameters using deterministic

elements. Then, we evaluate the compression capabilities of such approach on image and speech classification tasks with model architectures ranging from 14K to 26M parameters. Finally, we propose a hardware-based model weights generator suitable for FPGA-based single computation engines and, demonstrate that on-the-fly models outperform state of the art structured pruning methods in settings with restricted memory bandwidth.

Q2. Several algorithms for convolutions haven been proposed to better utilize on-board resources, inducing higher parallelism. They can directly be used irrespective of the data type, since these only perform a replication in memory of the convolution operands. This is not the case for some theoretically faster algorithms that operate in a transformation space, rendering them unusable due to numerical errors. *Can we capture such errors during training to learn models that are aware of such inherent limitation but remain accurate? Can we achieve so for Winograd convolutions, the fastest known algorithm for convolutions?*

In Chapter 4, this thesis addresses the problem inherent to current Winograd convolution implementations: severe numerical degradation in reduced precision contexts, which renders them unusable for light-weight deployments making use of integer arithmetic. We address this limitation by introducing a relaxation on how Winograd convolutions are implemented and, present an end-to-end Winograd-aware formulation of convolutional layers. To find the optimal convolution algorithm configuration for a given model we propose `wiNAS`, a neural architecture search framework that jointly optimizes a given macro-architecture for accuracy and latency leveraging Winograd-aware layers and a latency model. We demonstrate $1.5\times$ speedups compare to INT8 `im2row` (one of the most widely used optimized convolution implementations) with marginal accuracy degradation on Arm mobile CPUs.

Q3. As ML matures, it is being adopted in radically different applications and settings. However, some of the most well-studied optimizations, such as the use of reduced precision arithmetic for inference acceleration, has only been considered in the context of data structures over regular grids such as images, text or audio. One of such new domains are graphs, which are irregular grids. *What are the unique challenges that arise when quantizing graph neural networks? How do the different types of layers found in the literature perform under reduced bit-width given the same input graph?*

In Chapter 5, this thesis explores the viability of training quantized GNNs, enabling the usage of low precision integer arithmetic during inference. We perform a systematic study on how existing quantization approaches commonly used for CNNs and language models can be adapted for GNNs. Then, we introduce a stochastic topology-aware re-formulation

of quantization-aware training in which high degree nodes are quantized less frequently than lower degree nodes. In this way, accurate gradients flow more often through high degree nodes, impacting a larger neighbourhood and, improving overall training quality. In addition to the model size reduction and faster inference, using lower bit-widths indirectly helps to alleviate the data locality problem, another of the challenge that arises with GNNs. With the proposed framework, INT8 models perform as well as FP32 models in most cases and enables up to $4.7\times$ speedups on CPU.

In addition to the aforementioned chapters containing the contributions of this thesis, Chapter 2 introduces the relevant background for the works presented in this dissertation. It provides an overview on network optimization techniques such as quantization, pruning and sparse methods. Then it introduces exiting algorithms used to speedup convolutions, an overview on neural architecture search and, a discussion on data movement costs and the use of high-level proxy metrics in ML workloads. Finally, a detailed introduction to graph neural networks is provided as well as a discussion on the challenges that arise when accelerating them. This thesis ends with Chapter 6, where we summarize the contributions presented and discuss future research.

1.2 Publications

The main contributions of this thesis have already been published at the following international conferences and workshops. When relevant, $\star$ is used to denote equal contribution:

[324] Vincent Tseng, S. Bhattacharya, **Javier Fernandez-Marques**, Milad Alizadeh, Catherine Tong and Nicholas D. Lane. *Deterministic Binary Filters for Convolutional Neural Networks.* In *Proceedings of the Twnety-Seventh International Joint Conference on Artificial Intelligence* (IJCAI), 2018.

[103] **Javier Fernandez-Marques** , Vincent W.-S. Tseng, Sourav Bhattacharya and Nicholas D. Lane. *BinaryCmd: Keyword Spotting with deterministic binary basis.* In *Conference on Machine Learning and Systems* (MLSys), 2018.

[333] Stylianos I. Venieris*, **Javier Fernandez-Marques*** and Nicholas D. Lane. *unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-fly Weights Generation.* In *IEEE International Symposium on Field-Programmable Custom Computing Machines* (FCCM), 2021.

[104] **Javier Fernandez-Marques**, Paul N. Whatmough, Andrew Mundy and Matthew Mattina. *Searching for Winograd-aware Quantized Networks.* In *Proceedings of Machine Learning and Systems* (MLSys), 2020.

[316] Shyam A. Tailor*, **Javier Fernandez-Marques*** and Nicholas D. Lane. *Degree-Quant: Quantization-Aware Training for Graph Neural Networks.* In *International Conference on Learning Representations* (ICLR), 2021.

Other publications I contributed to during my DPhil studies but that are not part of this thesis are listed as follows:

[11] Milad Alizadeh, **Javier Fernandez-Marques**, Nicholas D. Lane and Yarin Gal. *A Systematic Study of Binary Neural Networks' Optimisation.* International Conference on Learning Representations (ICLR), 2019.

[244] Akhil Mathur, Daniel J. Beutel, Pedro Porto Buarque de Gusmão, **Javier Fernandez-Marques**, Taner Topal, Xinchi Qiu, Titouan Parcollet, Yan Gao and Nicholas D. Lane. *On-device Federated Learning with Flower*. In 2nd On-device Intelligence Workshop (MLSys) 2021.

[278] Xinchi Qiu, Titouan Parcollet, **Javier Fernandez-Marque**s, Pedro P. B. de Gusmao, Daniel J. Beutel, Taner Topal, Akhil Mathur and Nicholas D. Lane. *A First Look into the Carbon Footprint of Federated Learning* 2021. Preprint arXiv:2102.07627

[110] Yan Gao, Titouan Parcollet, Salah Zaiem, **Javier Fernandez-Marques**, Pedro P. B. de Gusmao, Daniel J. Beutel and Nicholas D. Lane. *End-to-End Speech Recognition from Federated Acoustic Models*, 2021. Preprint arXiv:2104.14297

Appendix A.1 provides details about my contributions to each of the above publications.

Chapter 2

Background

In this chapter, we aim to provide readers with an overview on the background relevant to the work presented in this thesis. First, Section 2.1 presents the set of techniques commonly used to design lightweight DNNs, namely model pruning, sparsification and quantization; in Section 2.2, we discuss why high-level proxy metrics to compare ML workloads might be flawed in the general case; in Section 2.3 we cover existing implementations of the convolution algorithm commonly used to accelerate CNNs, this will be relevant for Chapter 4; in Section 2.4 we introduce the challenge of data movement and discuss its impact for the efficient processing of ML-workloads; and conclude this chapter by providing a primer on Graph Neural Networks and the challenges associated to their acceleration in Section 2.5.

2.1 Lightweight Network Designs

For the last decade, the ML community has seen an unprecedented growth on the number of model architectures [305, 173, 146, 319, 165, 192, 329, 381, 79, 88, 125], training techniques [191, 174, 158, 175, 320, 230, 420, 236, 58], datasets [209, 199, 75, 227, 359, 68, 18, 269, 342, 279] and, frameworks [272, 2, 56, 57, 395, 202, 114, 260] to facilitate research and deployment of ML-powered commercial applications. This rapid development, primarily driven by performance metrics such as accuracy, spawned the design of models with high representational power relying on millions— and even billions [43, 301]— of parameters with complex execution patterns. This hindered their deployment on off-the-shelf hardware as well as restricting the viability of new custom hardware designs, effectively limiting their usage on real-world applications and contexts. To address the challenge of lowering the compute, memory and, energy requirements associated with high-performing models, the research community has relied, primarily, on three sets of techniques: pruning,

which results in model compression and acceleration during inference by discarding individual weights or entire channels; sparsity, which by zeroing weights or blocks of weights, inference can be accelerated on supported hardware; and quantization, which offers model size savings by representing parameters with fewer bits and accelerates inference using lower precision arithmetic (e.g. INT8, binary). In the second half of the decade, model design, spanning the choice of layers, their order, inter connectivity, and, their parameterization, stopped being a manual process and has been, for the most part, automated ever since. Today, the adoption of the aforementioned optimization techniques can also be included into the search space and, automate their inclusion and parameterization under a single neural architecture search framework [46].

The remaining of this section presents each of these orthogonal optimization techniques in further detail and in the context of the work presented in this thesis. With the exception of quantization, a largely application-agnostic optimization technique, this section would primarily focus on vision tasks when providing background on pruning, sparsity and, neural architecture search.

2.1.1 Pruning

Pruning neural networks involves discarding parts of the model that are redundant and, in this way, obtain a smaller, lightweight model architecture more suitable for deployment on constrained settings with limited memory and compute budgets. Typically, pruning a model is a three-step process where: first, an (allegedly) over-parameterized model is trained; then, following a objective-based criterion the model gets pruned; finally, a fine-tuning stage is used to update the remaining parameters in the model. Often pruning and fine-tuning happens iteratively for a fixed number of rounds. The goal of pruning is to meet the desired model size and latency constrains for a given task and hardware, while preserving the performance metrics (e.g. accuracy) from the original model.

Early works on neural network pruning date as back as the late 80s and early 90s [261, 177, 143] but, just like other dimensions in the ML space, the popularity and sophistication of this model optimization technique has grown rapidly in the last decade. What follows is an overview of the current landscape of pruning methodologies and strategies.

2.1.1.1 Pruning Methods

Pruning techniques can be categorized as *unstructured pruning* [140, 256, 143, 141, 135], where individual weights or neurons are removed or, *structured pruning* [148, 151, 181, 400, 257, 353], where entire channels or layers are discarded.

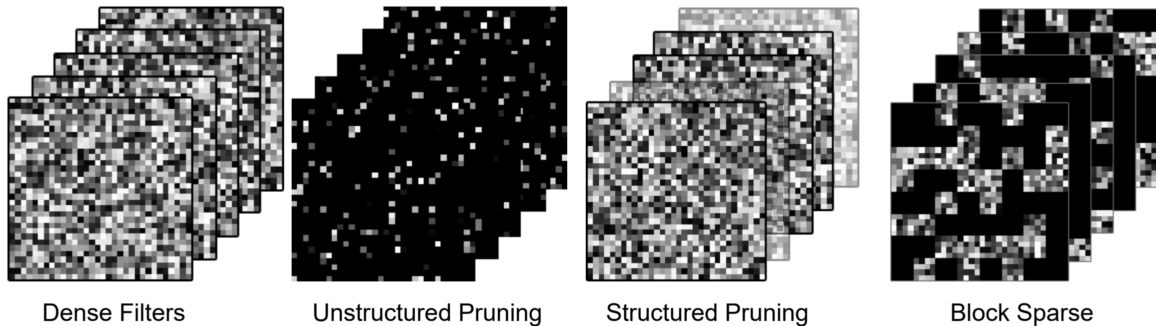


Figure 2.1: Illustration of the different types of sparsity often found on modern CNNs. Images follow a gray-scale color map where values containing zeros are shown in black. Unstructured sparsity can achieve very high compression ratios but is hard to accelerate on off-the-shelf hardware. On the other hand, structured sparsity can discard channels (shown with low opacity) to reduce the operations needed for inference, often resulting in lower latencies. Block sparsity sets tiles to zero while keeping others dense. Compared to unstructured sparsity, block sparse layers can be accelerated at lower ratios of overall sparsity on supported off-the-shelf hardware (e.g. Volta and Ampere NVIDIA GPUs)

Frameworks relying on unstructured pruning often achieve higher compression ratios at the expense of inference stages being as compute intensive in practice as those of the original model. This is because, assuming pruning has been homogeneously applied on the model, sparse operations can only be efficiently accelerated on supported hardware, such as modern GPUs [355, 401, 160] or custom accelerators [410, 238, 308], for a sufficiently high sparsity ratio. The lower the ratio, the less likely sparse operations would translate into measurable speedups. In the case of CPUs, speedups due to sparse operations where one operand is unstructurally sparse are often only realizable at 95% sparsity ratios or higher [356, 161]. Although this might seem like an unattainable[1] goal for vision or language models, for other forms of ML such as those operating on graphs, the high sparsity levels in the adjacency matrix guarantee the acceleration of certain stages during inference. Graph neural networks, however, have other challenges that limit their acceleration in practice, as it would be discussed later in section 2.5.2.

On the other hand, methods that apply structured pruning, trade model compression for acceleration potential. These approaches modify the underlying computational graph by discarding channels, resulting in smaller but still dense convolution operations, or by removing the nodes all together if an entire layer is set to be removed by the chosen pruning strategy. As a result, structured pruning frameworks are the preferred option when aiming to accelerate inference on general purpose hardware. A body of work across structured and unstructured pruning methods, attempts to induce structure in otherwise randomly sparse

[1]Early models such as VGG and AlexNet are known to be notoriously over-parameterized and, as a result, it is common to achieve very high sparsity ratios, i.e. above 90%, with marginal impact on accuracy [295, 140]

networks [291, 285, 364, 334]. This is often referred to as *block sparsity* and consists in subdividing the matrix representations of inputs or weights into tiles (e.g. 16×16 tiles), and restrict the training in such a way that some tiles contain only zeros while the rest remain dense and real-valued. Matrix-matrix multiplications following such a pattern can be accelerated at lower global sparsity ratios compared to those following unstructured sparsity [159]. Other forms of constraining how sparsity occurs have been proposed, for example, a cache-aware reordering on the sparsity pattern of the weights [96]. This can be used to ensure high cache reuse on Cortex-A mobile CPUs, resulting in $2.4\times$ acceleration of MobileNets. Each of the different pruning methods introduced so far are illustrated in Figure 2.1.

2.1.1.2 Pruning Strategies

Over the years, several techniques have been proposed to guide the pruning process. These include: discarding neurons based on their magnitude [140, 141], based on the weighted norm of channels [218], the sparsity level of the output feature map [167], or by randomly pruning channels [252], which can work as well as other heuristic-based pruning approaches. Others frame pruning as a minimization process where the task is to reduce the cross-similarity of the output and discard channels in the weights tensor accordingly [353, 312], or compute the contribution to the loss of each channel and inform an importance-based mechanism to prune these [257]. Other methods include high-level metrics such as FLOPS to guide the pruning process [128, 340]. However, these are only proxy metrics for latency and energy consumption [391, 149, 315], which might given a distorted view of the real savings only observable on real model deployments. In Section 2.2 we provide an expanded discussion on the problem of relying on proxy metrics.

A growing body of works present an alternative framing from the standard three-stage pruning process and, instead, propose pruning the model ahead of training. This can be achieved by identifying trainable sub-networks in dense models at initialization, as it was first formally framed in the Lottery Ticket Hypothesis [107]. The work of Lee et al. [212], prunes individual weights after learning a saliency-informed binary mask that preserves connections that minimize the difference in loss between the original model and the pruned one. GraSP [343] attempts to preserve gradient flows after pruning by removing weights that contribute the least to the gradient norm, prioritizing in this way the flowing of information during training. These two approaches perform unstructured pruning before training. More recently, the work of Amersfoort et al. [325] adapts SNIP [212] to the structured pruning setting, allowing also faster training stages without specialized hardware. While pruning before training can help providing insights on the role the network architecture

plays [107, 378] when excluding the weights, the quality of the resulting pruned models tends to be worse that that of models than followed the prune-then-fine-tune methodology. The latter, however, are yet to be systematically evaluated [39] to properly asses their performance and generalization. Pruning before training can therefore be seen as a more general approach since it is, for the most part, data-agnostic. It can also be labeled as a extremely lightweight form of, albeit very restricted, one-shot neural architecture search.

Until recently, it has been widely accepted that training an over-parameterised version of the model first and then prune it would yield better results than training the smaller architecture from scratch [48, 181, 400]. However, recent works demonstrate that, for a fixed number of non-zero parameters, sparse models tend to outperform their dense counterparts, even if these have more parameters [211, 187, 418]. This is an interesting line of future work which could lead to new types of architectures only suitable for very sparse models.

2.1.2 Quantization

Quantizing neural networks involves reducing the numerical precision of the operands and output of each layer in the compute graph describing the model. Unlike pruning, quantization preserves the layers and tensor shapes defining the model[2]. While training these models would normally be done using either FP32 of FP16—although recent approaches attempt at using integer-only arithmetic [349]—, enabling lower bitwidth model representations for inference, e.g. INT8, is the main goal as it enables deploying smaller models, with faster inference stages, lower energy consumption and requiring smaller chip area [315, 217]. Concretely, 8-bit quantizatized models have shown to achieve comparable performance to full-precision models [175, 198, 316, 347, 402, 274] while being ready for deployment on off-the-shelf hardware as 8-bit arithmetic is widely supported. In addition to resulting in a direct $4\times$ model size reduction, 8-bit integer-only arithmetic benefits from up to $116\times$ and $27.5\times$ chip area reduction compared to full precision additions and multiplies respectively, requiring $30\times$ and $18.5\times$ less energy [162, 365]. Because of these desirable benefits, 8-bit quantization has been widely adopted in both compute-constrained devices [224, 68, 348, 225, 99] and accelerators [366]. While lower precision networks exist and have been studied for a few years, for example binary [72, 228, 376, 234, 242] and ternary [215, 341, 76] networks, the challenges associated to training these and match accuracy of their FP32 or INT8 counterparts, haven driven the community to develop frameworks for 4-bit quantization [123, 28, 350, 264, 67] instead, which are easier to accelerate on commodity hardware.

[2] Although some layers, such as Batch-Normalization and some activation functions, can be fused to better accelerate inference. These techniques are discussed in section 2.1.2.2.

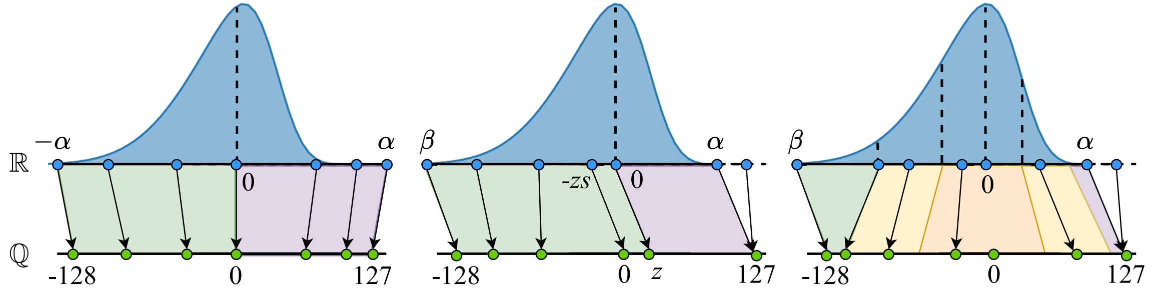


Figure 2.2: Illustration of the mapping between real-valued and quantization domains for three quantization implementations given a skewed Normal distribution: (right) uniform symmetric quantization; (middle) uniform affine quantization, which allocates more bits to the left side of the distribution; and, (right) a form of non-uniform quantization which allocates more bits to regions on the real line with higher concentration of values.

2.1.2.1 Quantization Implementations

The most widely used form of quantization is *uniform affine quantization*, which describes the quantization of a floating point vector $\mathbf{x}$ to its b-bits representation as:

$$\mathbf{x}_b = \texttt{clamp}\big(\lfloor \mathbf{x}/s \rceil + z, 0, 2^b - 1\big) \tag{2.1}$$

where scaling factor $s \in \mathbb{R}^+$ and zero-point $z \in \mathbb{Z}$ are used to map $\mathbf{x}$ to the range of representable values at the given bitwdith. Under this formulation the zero value is accurately represented, $\lfloor \cdot \rceil$ is the rounding operator and `clamp` is defined as:

$$\texttt{clamp}(\mathbf{x}, a, b) = \begin{cases} a & \mathbf{x} < a \\ \mathbf{x} & a \leq \mathbf{x} \leq b \\ b & \mathbf{x} > b \end{cases} \tag{2.2}$$

Then, the approximated real-valued of $\mathbf{x}$, $\hat{\mathbf{x}}$, can be obtained by reverting eq. (2.1):

$$\hat{\mathbf{x}} = s(\mathbf{x}_b - z) \tag{2.3}$$

where, in the context of enough numerical resolution (i.e. high bitwidth) and $\mathbf{x}$ being evenly spread on the $[0, 2^b - 1]$ range, $\hat{\mathbf{x}}$ and $\mathbf{x}$ would be indistinguishable. The overheads of affine quantization associated to the zero-point can be discarded by setting it to 0. This is known as *symmetric uniform quantization* and is often implemented by either mapping values to a signed, $[2^{b-1}, 2^{b-1} - 1]$, or unsigned, $[0, 2^b - 1]$, ranges. Another optimization involves restricting the scaling factor to be of the form $s = 2^{-f}$, where $f \in \mathbb{Z}$, enabling scaling $\mathbf{x}$ using bit-shifts. However, these optimizations can restrict the quality of mapping from $\mathbf{x}$ to $\mathbf{x}_b$, in particular in uneven or skewed distributions. Common to both affine and

symmetric quantization is the use of a fixed step size, the scale factor s, that defines the distance between representable values along the real line. This might result in a suboptimal mapping of values when many elements in $\mathbf{x}$ get mapped to the same quantized value while other quantized values remain unused. Intuitively a better approach would be such that assigns finer granularity in the range of values that concentrate the majority of values (e.g. around zero) and less granularity, i.e. wider bins, for the tails of a distribution. This is known as *non-uniform quantization* and several approaches have been proposed to define variable step sizes including logarithmic distributions [338, 16] or by treating it as a learnable parameter [186, 389]. Despite non-uniform quantization introduces lower quantization error, uniform quantization is often the better choice when considering the deployment of quantized models on off-the-shelf hardware due to its simplicity. Figure 2.2 illustrates the key differences between these quantization implementations.

2.1.2.2 Practical Considerations

In addition to the choice of quantization implementation, other design choices enable further control on the quality of models and ensure data is quantized as few times as possible:

- **Batch-norm folding.** A popular design choice in CNNs and other models is to make use of BatchNormalization (BN) [174] after convolutional layers to help achieve faster convergence rates and model generalisation. Like other layers, it gets inputs and generates outputs and, in the context of quantization, both should be quantized. To minimize quantization error, a common approach is to fold BN parameters into the preceding layer's weights and biases [198, 175].

- **Activation fusion.** Similarly to BN layers, being able to fuse the activation layer with the quantization stage of the preceding layer would result in one less quantization layer and its associated injection of quantization error. In some cases, this is easily achievable. For example, quantization layers followed by ReLU can be fused by setting the minimum representable value during quantization to zero. However, for other activations functions like Swish [280], Softplus (a smooth version of ReLU) or, Sigmoid, which perform non-linear mappings, fusing them is not possible.

2.1.2.3 Post-training Quantization

Frameworks making use of post-training quantization (PTQ) take existing FP32 models and convert them to b-bits models without retraining. The challenge is to calibrate the quantization ranges for weights and activations in order to minimize the degradation introduced by quantization. This calibration can be done taking the absolute min/max ranges

observed, use percentiles (e.g. clipping the top 0.1% of values) to discard outliers, by reducing the cross-entropy between an activations tensor $\mathbf{X}$ and its reconstructed version $\hat{\mathbf{X}}$ or, by reducing mean squared error between these. PTQ tends to work best with large over-parameterized models than with lightweight architectures (e.g. MobileNets), with choice of calibration method often being a function of the type of architecture and task at hand [372].

2.1.2.4 Quantization-aware Training

Although PTQ introduces minimal quantization error for some models, quantization-aware training (QAT) has become the *de facto* approach towards designing robust quantized models with low error at lower bitwidths [347, 402, 347].

In their simplest form, QAT schemes involve exposing the numerical errors introduced by quantization at training time by simulating this stage on the forward pass as in eq. (2.1) followed by eq. (2.3). Then, during backward pass, the straight through estimator (STE) [33] is used—as if no quantization had been applied. This is often called *fake quantization*. During training, the quantization layers are responsible for tracking the input tensor's min and max values, $\mathbf{X}_{\min}$ and $\mathbf{X}_{\max}$, which are then used to compute the zero-point z and scale s parameters. Two popular ways of performing such tracking are: $\min/\max$, which tracks the min/max tensor values observed over the course of training; and *momentum*, which computes the moving averages of those statistic. The latter introduces an averaging constant c, which is often set to 0.01. Instead of simply tracking such statistics, recent approaches propose learning z and s. This proved [97, 176] to result in better accuracy retention than other forms of QAT.

To reach performance comparable to FP32 models, QAT schemes often rely on other techniques such as *gradient clipping*, to mask gradient updates based on the largest representable value at a given bitwidth; noisy QAT, which stochastically applies QAT to a portion of the weights at each training step [309, 87]; or the re-ordering of layers [299, 11].

Quantization has been the object of study of a number of survey works [117, 372, 263, 198, 276]. We refer the interested reader to these for a more in depth analysis.

2.1.3 Automating Neural Architecture Search

Automating the process of designing neural network architectures has drawn considerable attention. The goal of NAS is to discover the best performing architecture given: a set of candidate layers or blocks of layers and, target application-specific metrics being accuracy and latency the most commonly used. In this way, the fundamental difference between the many NAS techniques proposed in the literature, is how those architectures are discovered.

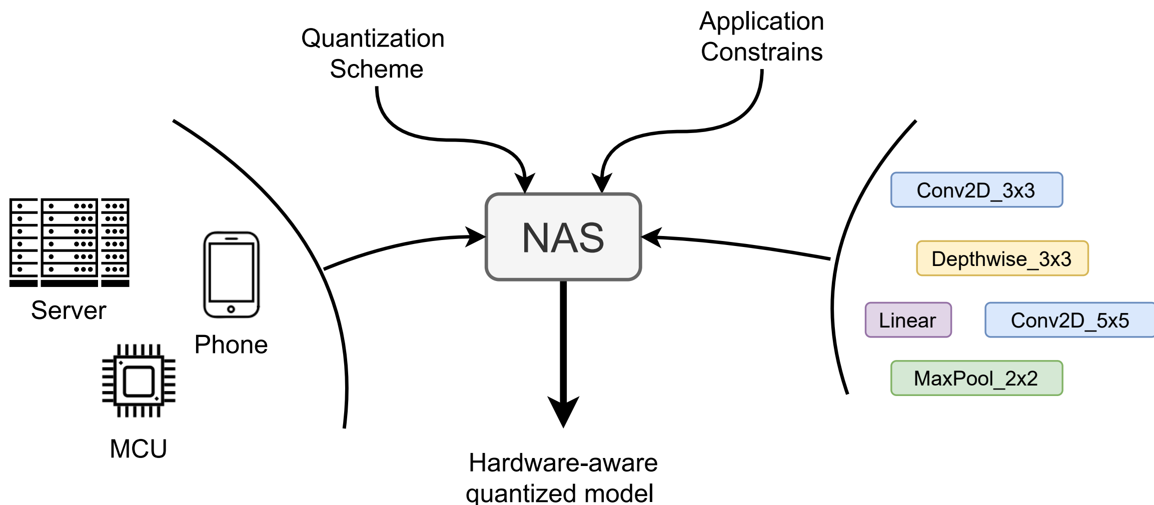


Figure 2.3: A generic hardware-aware NAS diagram. During search, the NAS framework could be guided by introducing several elements such as: latency models measured on the target platform or amount of on-chip memory; the search space can further restricted by introducing application-level constraints such as minimum accuracy or maximum energy consumption; similarly, the search space can be expanded by considering different quantization implementations, each offering different trade offs between attainable speedup, numerical degradation and software/hardware support.

Early attempts relied on reinforcement learning (RL) [420, 42, 283, 318] or Bayesian optimization [153, 100] and required thousands of GPU hours to converge due to their computationally expensive and exhaustive search stages. Other works opted instead for a gradient-based search by framing the problem as learning a single over-parameterized network, or *supernet*, where all *candidate operations* at a particular node (e.g. a layer) are taken into consideration. Once trained, an architecture can be sampled based on application-specific constraints such as number of parameters or FLOPs. These NAS paradigm is often referred in the literature as to *one-shot* NAS. The main aspects differentiating gradient-based NAS approaches are the procedure followed to sample the supernet and, the way the output of a layer combines the contribution of each candidate operation when training the supernet. While Bender et al. [32] combines them as the sum and DARTS [230] as a weighted sum, ProxylessNAS [47] relies on path-level binarization, (i.e. two candidate operations in each layer are considered per batch, then one gets its score increased while the other is decreased) making it possible to perform the search on the entire architecture directly using a single GPU. This effectively reduces the memory needs during training to that of training a standard model. More importantly, and unlike prior approaches which generally perform the search in a proxy space, ProxylessNAS enables the training of the entire architecture directly. This formulation also enables the introduction of other per-layer metrics such as latency. In Chapter 4, we rely on this key property of ProxylessNAS to identify the most

suitable convolution algorithm for each layer in a fixed model architecture.

NAS has been predominantly used for vision tasks such as image classification or object detection. On the other hand, its adoption in other domains such as for natural language processing (NLP), automatic speech recognition (ASR) or, for discovering new graph neural network architecture [111], has been slow. Nonetheless, the imminent adoption of NAS as an indispensable tool for all these tasks led to the introduction of several NAS benchmarks. These not only restrict the search spaces with the types of layers or blocks that are known to work well for the application at hand, but also provide performance metrics for all the possible models constructed from the search space. Some of the popular benchmarks are NAS-Bench-101 [397], NAS-Bench-102 [86] which extends the former, NAS-Bench-NLP [194] and, NAS-Bench-ASR [248].

In addition to architecture discovery, NAS has also been successfully used for automated network pruning [150] and quantization [348] to identify hardware-aware designs (this is illustrated in Figure 2.3), facilitate hardware co-design of FPGA-based accelerators [3], chip floorplanning [1] and placement [250], both using RL.

2.2 On Proxy Metrics and Lower-level Optimizations

Regardless of the optimization goal (e.g. compression, acceleration, lower memory peak, etc), all of the previously introduced techniques impose a trade-off between efficiency during inference and output quality. In order to compare such trade-off against other approaches that follow a similar or alternative optimization strategy, high-level metrics such as number of model parameters or number of FLOPS during inference have been widely used. However, these metrics might only loosely correlate with the *real* memory consumption and latency observed upon model deployment. For example, model size does not account for memory utilization, including memory peak, needed to have the layer input and output buffers in memory. With the following example we aim to show how easy it is to construct a convolutional layer with almost identical number of parameters but outputting tensors of very different shape. Given a 32×32 RGB input and two convolutional layers: `conv2dA` with stride two, kernel size of 7 and 32 output channels; and `conv2dB` with stride one, kernel size of 3 and 160 output channels. Although both roughly have the same number of parameters, `conv2dB` outputs a tensor that is $20\times$ larger. This evidences that model size alone is not a good proxy metric for memory utilization.

When it comes to estimating the latency of a network, FLOPS have been a widely used proxy metric. However, it has been shown [370, 84] that FLOPS often do not correlate with

real latency. A similar example to the previously given for model size can be constructed to show networks with higher FLOPS can run faster than others with fewer operations. In Sandler et al. [293], a MobileNetV1 with 575M FLOPS is shown to be $1.6\times$ faster on a Google Pixel 1 than a NasNet-A [421] that require 564M FLOPS. These differences are even more dramatic if making the comparison between two radically different architectures: a EfficientNet-B4 [319] requires 4.2GFLOPs compared to the 4.5GFLOPs needed for a visual transformer (DeiT-Small) [322, 130]. Both achieve over Top-1 82% on ImageNet but the latter is $15\times$ faster on an server-grade Arm-Graviton2 CPU. The reason for this large discrepancy in observed latency is due to the *hidden costs* of deploying often complex compute graphs incurring into sub-optimal mapping of operations (e.g. `matmul`) onto the available hardware, poor data locality for models with multi-branch architectures, order of operations or, large activations that require frequent accesses to main memory on devices with small caches. A better proxy than FLOPS consists of summing up the measured latencies for each individual layer in order to obtain an overall latency estimate [47]. However, this methodology still might fail to capture the impact of the last aforementioned hidden costs. Dudziak et al. [90] have shown that predicting latency can be done by means of a graph neural network, which better captures the non-linear mapping between FLOPS and latency for a variety of platforms.

While graph-level changes in the compute graph describing the model (as those obtained with structured pruning) can be an effective way at reducing operations and memory utilization, other lower-level optimizations with regards to how each operator maps to the available hardware or how memory buffers are re-utilized can provide further speedups. To attain such optimizations, one must seek them closer to the metal and, generally at compile time. TVM [57] is a compiler for deep learning targeting CPUs, GPUs and, specialized accelerators. At its core, it exposes the graph-level and operator-level optimizations at compile time, enabling the same compute graph to be seamlessly ported to a wide variety of platforms. Graph-level optimization include *operator fusion* which mergers consecutive layers into a single one (e.g. `Conv2d+BN+ReLU` fused into a single operator), reducing memory accesses. On the other hand, node-level or operator-level optimizations include data *layout transformations* that make better utilization of vector (e.g. SIMD) and tensor (e.g in modern GPUs and TPUs) compute units, or automate the selection of specialized implementations of the same operator (e.g. convolution algorithms) through built-in benchmarking routines. TVM also has support for optimizing ML workloads on custom hardware and can be extended to add expert knowledge to better capture certain properties of the hardware or the compute graph at hand.

2.3 Fast Convolution Algorithms

In addition to the use of lightweight CNN layers such as depth-wise convolutions [165, 293, 164], the use of integer arithmetic or sparse operations, among other techniques, inference can be further accelerated by making use of one of the multiple and widely supported ways of performing the convolution operation.

The standard approach to computing multi-dimensional convolutions is to *lower* the operation to a matrix multiplication, which can be conveniently implemented using heavily optimized General Matrix Multiplication (GEMM) libraries. Hence, different implementations of Multiple Channel Multiple Kernel (MCMK) convolutions have been described: the `im2` family of algorithms, such as `im2col` [179, 131], re-arrange (by replication) the operands in memory enabling higher throughput than the textbook sum-of-single-channels algorithm; the `kn2` algorithms, e.g. `kn2row` [328], which alleviates the memory needs of `im2row` by leveraging GEMM primitives applied to the layer's input directly. This however comes at the cost of very limited striding options without incurring into substantial data movement penalties. More recently, Indirect Convolutions [92] proposes the introduction of indirection buffers as a counter measure to avoid explicit data replication needed in `im2col`-based implementations.

While the previous methods treat the textbook definition of convolution as a matrix-matrix multiplication, alternative formulations of the convolution operation first envisioned by the signal processing community have also been adapted to CNNs. These include: the use of FFTs, which replace convolution with its multiplication-only counterpart in the frequency domain resulting in faster inference [243, 6] and training [155]; the Strassen algorithm [310], which when applied to convolutions [69, 323] significantly reduces the number of multiplications at the cost of more additions; or the Winograd algorithm [367], which replaces convolutions with a set of matrix transformations and point-wise multiplications and, results in significantly faster inference stages [207].

All these different implementations of the convolution operation come with their own set of pros and cons: *direct-loop* (i.e. the standard sliding window approach) offers minimal memory overheads but often becomes the slowest option; `im2` are generally faster but results in memory overheads due to the need of replicating the input prior to perform the GEMM; Winograd is widely accepted as the fastest algorithm, comes with manageable memory overheads but suffers from numerical degradation, leading to its exclusion for deployments using quantization; and, FFT which, although shares a similar structure as Winograd, but cheaper $\mathcal{O}(n \log n)$ transforms of inputs and weights, it only translates into

speedups when using larger kernels, much larger than the 3×3 or 5×5 often found in modern CNNs. Furthermore, FFTs perform the element-wise multiplication of complex-valued operands, requiring additional compute compared to real-valued multiplies [91, 188]. In practice, the use `im2` algorithms are the default choice when deploying CNNs and only *direct-loop* is considered in those scenarios where memory and memory bandwidth are limited, as is the case of low-end microcontrollers.

2.4 The Data Movement Challenge

For ML workloads, the term *data movement* is often used to collectively denote the processes of: reading model parameters and layer inputs from main memory; fetching them from the various levels of cache into the registers; writing the layer output to main memory, to the lower-level cache or, to both. The complexity of analyzing data movement is tightly coupled with the hardware and the compute graph representing the ML model. For CPUs, data movement is managed by the OS, restricting the amount of optimization that can be achieve along this axis, although some works have proposed cache-aware regularization mechanisms during training to facilitate higher cache hit rates [96]. In the case of MCUs, which often come with a single level of cache or no cache at all, this analysis gets substantially simplified also in part to their single-threaded nature. On the other hand, platforms such as FPGAs require an explicit management of data movement including the parameterization of on-chip buffers (i.e. their size and routing). These buffers act as caches for reads and partial layer outputs. Such full control of on-chip memory hierarchy (as well as compute resources) enables to design custom solutions for a particular ML workload sustaining throughput levels that are unattainable by off-the-shelf hardware under the same power envelope.

In addition to the latency costs associated to the accessing of lower-level memories (e.g. DRAM but also Flash drives), these accesses also require vastly different amounts of energy. For example, retrieving the same amount of data from DRAM compared to from an L1 cache requires over two orders of magnitude more energy. By going one step up the memory-hierarchy ladder to a L3 SRAM cache, this gap is reduced to about $6\times$ [315]. We now provide further context and a concrete example: under the 45nm CMOS fabrication process an FP32 add consumes 0.9pJ, a 32bit SRAM cache access requires 5pJ, while the same access to DRAM takes 640pJ. Accessing DRAM requires over $700\times$ more energy than performing a FP32 addition [162, 140]. This evidences that energy consumption can easily be dominated by memory accesses, specially if models do not fit in system memory. Easing the costs of intra-device data movement is therefore of paramount importance when

deploying ML-powered applications on commodity devices running on batteries. Because of the complexity associated to controlling data movement outside custom hardware implementations such as FPGAs, the community has indirectly addressed the problem of data movement by proposing lightweight architecture designs and quantization.

While intra-device data movement has attracted the most attention, inter-device data movement also presents a similar distance-energy relationship and challenge. Even when relying on low power communication mechanisms such as BLE or ZigBee for close-range communications, energy consumption remains in the order of hundreds of milliwatts [303] when transmitting data. On the other hand, and unlike displays and radios, CPUs and sensors require considerably less power: real-time audio processing at 384 KHz requires between 10-25 mW [156] using a Cortex-M4; and, a low-resolution image sensor suitable for object tracking applications consumes 277μW [290].

The work presented in this thesis, with a bigger emphasis in Chapter 3, considers intra-device data movement costs. For newer forms of ML, such as Federated Learning, which could be viewed as a form of distributed learning where nodes are commodity devices such as smartphones or other smart devices, both intra-device and inter-device data movement costs should be considered.

2.5 Graph Neural Networks

Most Graph Neural Networks (GNNs) may be viewed as generalizations of CNN architectures to an irregular domain: at a high level, graph architectures attempt to build representations based on a node's neighborhood. However, unlike CNNs which operate on regular grids (e.g. images), a node's neighborhood does not have a fixed ordering or size. Because GNNs operate on irregular topologies, i.e., graphs, a number of challenges arise when attempting to accelerate these workloads. Nevertheless, GNNs are the preferred option when it comes to modeling irregularly structured data in applications as diverse as molecular interactions [94, 374], social networks [137], recommendation systems [326, 419] or program understanding [12, 396, 265]. This section first provides an introduction on how GNNs work and, presents some of the most popular layers often used to build GNNs. Finally, we highlight the three major limitations that make GNN acceleration challenging.

2.5.1 A Primer on Graph Neural Networks

A graph $\mathcal{G} = (V, E)$ has node features $\mathbf{X} \in \mathbb{R}^{N \times F}$, an incidence matrix $\mathbf{I} \in \mathbb{N}^{2 \times E}$, and optionally D-dimensional edge features $\mathbf{E} \in \mathbb{R}^{E \times D}$. The incidence matrix $\mathbf{I}$ is often transformed into its adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ equivalent.

Early works for GNNs focused on spectral representations for graphs in the Fourier domain, involving the learning of filters with various levels of spatial locality [44, 74, 152]. Under this formulation, signals on graphs are filtered based on the eigendecomposition of the graph Laplacian [302], which encodes the graph topology and nodes' degree. This dependence on the graph Laplacian is one of the limitation of spectral methods, which assume a fixed graph topology, making them unsuitable for applications where the learnt GNN model needs to generalize to unseen graphs (e.g. as is the case in graph classification or graph regression tasks) or consume dynamic graphs (e.g. point clouds).

In this chapter we consider GNNs architectures operating on the *spatial* domain, which apply the convolution operation directly on the graph, by aggregating nodes' neighbourhood information. This is done by adapting the notion of standard convolutions, that normally operate on regular grid (e.g. images), to an irregular grid. This can be formulated conforming the *message passing* paradigm for neural networks (MPNN) [118], which represents the inference stage in GNNs as a three step process:

- **Message generation:** This stage involves the transformation of node features $\mathbf{h}^l$ in a given layer l in the GNN (recall that $\mathbf{X} = \mathbf{h}^0$) using $\mathbf{W}^l \in \mathbb{R}^{F \times F'}$, the matrix representing the layer parameters that perform a projection from F to F' dimensions. The new feature embeddings can be expressed as $y^l = \phi(\mathbf{h}^l, \mathbf{W}^l)$, where ϕ denotes differentiable functions such as MLPs (Multi Layer Perceptrons). The most common approach is to perform a matrix-matrix multiplication $\mathbf{y}^l = \mathbf{W}^l \mathbf{h}^l$. Then, the node messages are constructed by applying an architecture-specific normalization mechanism (e.g. degree-based or attention-based normalization) or left unnormalized.

- **Message aggregation:** In the aggregation stage, each node gathers and aggregates the messages, $\mathbf{y}^l$, of each neighbouring node. More formally, the aggregation of messages at node v_i in layer l is defined as $o_i^l = \mathbf{y}_i^l + \square_{j \in \mathcal{N}(i)}(\mathbf{y}_j^l)$, where $\square$ denotes a differentiable, permutation invariant function (e.g. sum, mean) and $\mathcal{N}(i)$ represents the set of nodes that belong to v_i's neighbourhood.

- **Feature Update:** The final stage involves passing the output of the aggregation through a nonlinearity γ (e.g. ReLU) and obtain the activations before proceeding to the next layer of the network: $\mathbf{h}^{l+1} = \gamma(o^l)$.

Putting everything together and, including the use of edge features, we can represent the process of updating the node features of node v_i at layer l as:

$$\mathbf{h}_i^{l+1} = \gamma\big(\mathbf{h}_i^l, \square_{j\in\mathcal{N}(i)}\phi\big(\mathbf{h}_i^l, \mathbf{h}_j^l, e_{i,j}\big)\big) \tag{2.4}$$

The majority of GNNs presented in the recent literature can be expressed with this formulation, each making use of different ϕ, $\square$ or γ as well as introducing various message normalization mechanism. Here we consider some of the most common layer architectures for GNNs. These are also the building block for more sophisticated networks:

- Graph Convolution Network (GCN) [192]: Arguably one of the simplest form of convolutional GNNs, a GCN updates node features by performing a degree-based weighted sum over the messages in a node's neighbourhood. This formulation is a linear approximation of spectral convolutions:

$$\mathbf{h}_i^{l+1} = \gamma\left(\sum_{j\in\mathcal{N}(i)\cup\{i\}}\left(\frac{1}{\sqrt{d_i d_j}}\mathbf{W}^l\mathbf{h}_j^l\right)\right) \tag{2.5}$$

  where d_i refers to the degree of the i-th node and γ represents a ReLU function.

- Graph Attention Network (GAT) [329]: By introducing a self-attention mechanism (e.g. a single-layer MLP), nodes attend differently to each of their neighbouring nodes based on the alignment of their respective feature vectors. This is one example of anisotropic GNNs. The updated node features using a single attention head can be expressed as:

$$\mathbf{h}_i^{l+1} = \gamma\left(\alpha_{i,i}^l\mathbf{W}^l\mathbf{h}_i^l + \sum_{j\in\mathcal{N}(i)}\big(\alpha_{i,j}^l\mathbf{W}^l\mathbf{h}_j^l\big)\right) \tag{2.6}$$

  where α represent the soft-maxed attention coefficients. In practice, $\mathbf{K}$ attention heads are used, each attending to a different projection in the node feature space. For such multi-head attention scenario, the above formulation becomes a concatenation of $\mathbf{K}$ terms and, the non-linearity γ is applied at the end.

- Graph Isomorphism Network (GIN) [381]: The graph isomorphism problem asks a simple question: *Are graphs $\mathcal{G}$ and $\mathcal{G}^\star$ topologically identical?* The Weisfeiler-Lehman (WL) test is a way to answer this question [363]. For certain graph topologies and aggregation functions $\square$, GNNs such as those using GCN layers fail the WL test. Such observation motivated the introduction of a generic formulation for GNNs where the result of the aggregation gets transformed by means of MLP stage:

$$\mathbf{h}_i^{l+1} = \text{MLP}^l\left(\big(1+\epsilon^l\big)\mathbf{h}_i^l + \sum_{j\in\mathcal{N}(i)}\mathbf{h}_j^l\right) \tag{2.7}$$

where ϵ is a learnable constant. The discriminative power of GIN networks is equal to the power of the WL test and, as a result, more theoretically suitable for graph-level tasks such as graph classification or graph regression.

- Residual Gated Graph Convolution Networks (GatedGCN) [40]: This layer is one of the first (and very few) making use of edge features and, updating them during training at each layer. In its most generic formulation, GatedGCN learns an edge-informed dense gating mechanism, effectively bringing full anysotropy to an otherwise GCN-like architecture. An attention tensor, $\eta_{i,j}$, is learnt at each layer and is used during message aggregation to weight each dimension of the node features:

$$\mathbf{h}_i^{l+1} = \gamma\Big(\mathbf{U}^l\mathbf{h}_i^l + \sum_{j\in\mathcal{N}(i)} \eta_{i,j}^l \odot \mathbf{V}^l\mathbf{h}_j^l\Big) \tag{2.8}$$

  where $\odot$ represents the Hadamard product, $\eta_{i,j}^l = \sigma(\mathbf{A}^l\mathbf{h}_i^l + \mathbf{B}^l\mathbf{h}_j^l)$ and, matrices $\mathbf{U}^l, \mathbf{V}^l, \mathbf{A}^l, \mathbf{B}^l \in \mathbb{R}^{F\times F}$ are layer parameters. It remains an open research question what is the appropriate way of incorporating edge features to the learning of GNN models. The optimal approach would likely be application dependent.

- Edge Convolution Networks (EdgeConv) [352]: Point cloud data can be interpreted as graph as long as its topology, i.e. incidence matrix $\mathbf{I}$, is defined either as a pre-processing stage or at run time. This is a challenging task, not only because of the associated compute and memory challenges of operating with large point clouds, but because defining a metric for node self-similarity and inform the process of creating edges is highly task dependent and remains an active research area [85, 369, 288]. EdgeConv layers present a way of constructing edge features as a concatenation of local and neighbourhood information by leveraging the edges following a k-NN stage:

$$\mathbf{e}_{i,j}^l = \mathrm{ReLU}\Big(\Theta^l\big[\mathbf{h}_i^l \parallel \mathbf{h}_j^l - \mathbf{h}_i^l\big]\Big) \tag{2.9}$$

  where Θ^l are trainable parameters. Then, the new node features are obtained following a max aggregation stage, $\mathbf{h}_i^{l+i} = \max_{j\in\mathcal{N}(i)}(\mathbf{e}_{i,j}^l)$. Interestingly, the authors propose recomputing the graph connectivity after each layer, demonstrating that the connectivity in deeper layers become more semantically meaningful than maintaining the Euclidean-based connectivity from the input graph.

A GNN layer, as those just introduced and many others present in the literature [371, 259, 320, 193, 78, 317, 36, 330], specifies how node messages are generated, aggregated, updated and, it generally allows for different aggregation strategies, i.e., choices of $\square$.

While `sum`, `min/max` or `mean` have become the most common aggregators in the literature, recent works put the focus on this often overlooked operator, $\square$ and, propose new methods that better discriminate node-node neighbourhood interactions. Two examples of such approaches are: Principal Neighbourhood Aggregation(PNA) [70], where the results of four simple aggregators are combined and independently scaled based on the central node's degree to generate the final contribution of each message; or the work of Wang & Karaletsos [351] where the aggregation stage is framed as a stochastic process in which noise is added to the edges, effectively re-weighting the contribution of each node. This apparently subtle change in the formulation proves to be helpful in alleviating the over-smoothing problem that arises when designing deeper GNNs.

2.5.2 Efficient Processing of Graph Neural Networks

Graph Neural Networks represent a new computation paradigm that, as introduced in Section 2.5.1, significantly differs from other deep learning workloads such as those relying on CNNs. Although some stages in the GNN message-passing pipeline share a similar data-flow as other types of networks operating on regular data (e.g. images, text, audio), where similar optimizations can be applied, other factors unique to graph data make the efficient processing of GNNs challenging. The current landscape of applications making use of GNNs face the following challenges:

Alternate execution patterns. GNNs are comprised of two very distinct types of workloads: a node-level workload, during the message generation stage, in which node features are transformed to a new embedding space; and a graph-level workload, during the message aggregation stage, in which nodes aggregate the new representations of neighbouring nodes. The first type of workload is similar to those found in CNNs or language models as it characterizes by regular data pattern accesses, high levels of data reuse (i.e. high cache hit rate) and, being compute bound [386, 59]. As a result, this stage benefits from software and hardware level optimization used for the aforementioned type of architectures. On the other hand, accelerating the graph-level workload requires a different set of optimizations since such stage is notoriously memory-bound as it is dominated by up to $\mathcal{O}(E)$ irregular memory accesses. This could lead to over an order of magnitude lower L2 cache hit rate than node-level workloads [386, 133], translating also into higher latencies and energy consumption. To address these challenges, some works pre-process the input graph by: reordering the nodes in order to maximize neighbourhoods overlap [362] and, cache hits as a result; identifying node communities and map those that are more densely connected

to caches closer to the compute [17]; reordering and sharing of intermediate node computations [59]; identifying and removing redundant edges [406, 404]; and, making use of custom hardware blocks to accelerate scattered memory accesses [387, 115, 21].

Large activations. Graphs, in particular those from social networks and recommender systems, can easily contain millions of nodes and close to a billion edges. As a result, and despite GNNs models containing significantly fewer parameters than large CNNs, the number of operations needed during inference is several orders of magnitude higher, as it is illustrated in Chapter 5's Figure 5.1. Inference and specially training on large graphs is therefore only possible on systems with large amounts of memory. Even server-grade GPUs with VRAM amounts approaching 100GB per card might not be able to hold high-dimensional activations for each layer in the network. An effective way of reducing memory usage is by sampling the graph at each step and operate on mini-batches of smaller sub-graphs. Over the years, a number of frameworks have been proposed: at each layer and training step, GraphSAGE [138] samples at most K neighbour nodes from each node, effectively delaying the *neighbourhood explosion* problem[3] and enabling the generation of node embeddings for inductive settings such as evolving graphs or entirely new graphs not seed at training time. FastGCN [53] performs importance-based node sampling at each layer instead of sampling nodes' neighbourhoods, ensuring in this way that the number of nodes remains constant at each layer. This method proved to be over an order of magnitude faster than GCN-based GraphSAGE on the Reddit dataset. Unlike the previous two methods, and many others in the literature [171, 398, 54], GraphSAINT [405] constructs mini batches by sampling the graph directly, as opposed to performing the sampling at each layer, using a community-aware sampling mechanism, leading to faster training convergence and better performance. Another approach building mini-batches from sub-graphs is ClusterGCN [63], which uses an off-the-shelf clustering algorithm (e.g. METIS [189], Graclus [80]) to partition the graph ahead of training, in this way the neighbourhood expansion in deeper layers is confined within each nodes' cluster and therefore allowing the training of deeper GNNs without incurring into memory issues. These sampling techniques are orthogonal to graph reordering optimizations to reduce on-chip cache utilization. For example, latency in ClusterGCN could be further reduced by ensuring the nodes in a cluster are contiguous in memory, promoting coalesced memory accesses, and, a better use of the cache hierarchy.

Approaches for implementing message propagation. A general and well studied formulation for implementing this stage is by means of `gather/scatter` operations [105,

[3]This refers to the scenario that arises when training deep GNNs in which, nodes in upper layers get their gradients after backpropagating through a (potentially) exponentially increasing number of nodes.

144, 124, 208], which are also the building blocks for other applications beyond GNNs. During `gather`, messages from nodes in neighbourhood of central node v_i are fetched, likely from scattered locations in system memory, and copied into a new intermediate matrix, $\mathbf{G_i} \in \mathbb{R}^{d_i \times F}$, where d_i is v_i's degree. This operation would normally be performed on all nodes simultaneously, which leads to an intermediate memory footprint of $\mathcal{O}(E)$, $\mathbf{G} \in \mathbb{R}^{E \times F}$, where E is the number of edges in the graph. This materialization of messages is therefore a major concern in larger or dense graphs. The result of `gather` ensures messages to a given node are contiguous in memory, facilitating the independent scaling of messages and applying the aggregation function, $\square : \mathbf{G} \in \mathbb{R}^{E \times F} \mapsto \mathbf{S} \in \mathbb{R}^{N \times F}$. Then, $\mathbf{S}$ is given to the `scatter` routine to update each node's representation. GAT is a layer that needs explicit materialization of messages to compute the attention coefficients. An alternative formulation to the use of `gather/scatter` is performing both the message fetching and aggregation computation as a single sparse matrix-matrix multiplication, `spMM`, between the adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and the messages $\mathbf{Y} \in \mathbb{R}^{N \times F}$. This is the preferred approach for architectures such as GIN, that do not require scaling of node features, or GCN, where messages are inversely scaled by the product of degrees along the edge (effectively becoming a one-dimensional edge feature). For the latter, the degree-based scaling factor can be embedded in $\mathbf{A}$ before multiplying with $\mathbf{Y}$. While many of the popular GNN layers can make use of `spMM` for aggregation, avoiding the burden of materializing messages and achieving speedups on supported hardware [4], it remains a challenge to efficiently make use of `spMMs` in the context of graphs with bidirectional data flows, multi-dimensional edge features or, message-based anisotropy mechanisms. For those types of graphs, more efficient implementations of `gather/scatter` might be required.

2.6 Summary

As discussed in this chapter, the now maturing field of ML, in particular when it comes to CNN-based architectures, has welcomed a plethora of optimization techniques ranging from quantization, to pruning and, from lightweight architecture designs to fast convolution algorithms. These have not only contributed to design larger, better performing models but also made it possible to deploy them on constrained devices such as smartphones and microcontrollers. With regards to GNNs, the field is steadily advancing as it draws interests from core-ML and systems communities to support new types of applications and address

[4] For example `cuSPARSE` on modern NVIDA GPUs.

some of their inherent challenges. By reviewing the recent literature we also identified three limitations and opportunities that motivate the research presented in this thesis:

1. **Compression for model acceleration.** Structural pruning, quantization and to some degree, sparsification, have dominated the efforts when it comes to accelerating inference. This set of techniques fixate on reducing the number of operations by removing redundant model elements, reducing the numerical precision of operations, or both. This translated as well into smaller models sizes which contributed towards easing the data movement problem. However, this benefit is often either not acknowledged or never treated as a first-class optimization dimension, despite it being one of the main sources of on-device energy consumption [315]. In Chapter 3, we present a new formulation for compression frameworks that achieves acceleration by reducing off-chip memory accesses, with model parameters being generated on-the-fly following an efficient reconstruction process.

2. **Convolutions in transformation spaces.** Convolution algorithms that operate in transformation spaces require special attention when it comes to their implementation using integer-only arithmetic, as these involve several stages all of which should make use of quantization. This is not the case for other algorithms such as `im2row`, since they operate directly on the input and weights tensors or, replicated versions of these. This made them a good choice when considering deploying quantized models. In Chapter 4, we address the problem of using quantization along with the Winograd algorithm, which operates in a transformation space (the Winograd domain), and propose a solution to other limitations it is known to suffer from.

3. **GNNs are dually bounded.** GNNs are notoriously difficult to accelerate, primarily due to them having two very distinct execution patterns during inference, each requiring a different optimization strategy. In Chapter 5, this thesis challenge this assumption and, present a framework that enables the learning of quantized GNNs, alleviating both the compute-bound node-level and memory-bound graph-level workloads while retaining good performance on various tasks. Chapter 5 includes the first published characterization of the challenges that arise when quantizating GNNs.

Chapter 3

On-the-fly Weights for ML Acceleration

Compressing ML models is desirable because it allows for their deployment on constrained devices, where memory and on-chip storage come at a premium. In addition to the compute needed during inference, data movement has a large impact on both latency and energy consumption, specially if memory accesses are predominantly done to DRAM or to lower levels in the memory hierarchy such as flash storage. Motivated by the urge to deploying ML-powered applications on commodity devices, considerable efforts have been devoted to designing more compact models. Among the various approaches presented in recent years, those performing unstructured compression are the ones that achieve the highest compression ratios. These frameworks often rely on lossy (e.g. K-means clustering, SVD) and lossless mechanisms (e.g. Huffman encoding, sparse formats such as CSR or COO) applied during training or as a post-training stage. The result is a extremely compact model representation with, hopefully, marginal degradation on performance. A well known example is DeepCompression [140], capable of compressing AlexNet by $35\times$, from 240MB down to 6.9MB with no accuracy loss.

Although compressing a model eases the deployment to devices across the network and, enables the storage of multiple models in devices with limited storage, most of unstructured compression frameworks require either specialized hardware to achieve measurable acceleration or a ”decompression” stage prior to doing inference, the latter effectively voiding most, if not all, acceleration potential. Such stage involves undoing the compression steps used to derive a compact model: reverting Huffman encoding, the sparse weights representations or, the clustering of parameters. These stages necessarily translates into additional latency and memory overheads during inference. Moreover, in those scenarios where the fully decompressed model cannot be materialized due to memory constrains, the decompression process needs to be repeated for each new input to the network, severely slowing down inference. Because of these reasons, the community favored instead other techniques,

namely quantization and lightweight architectural designs, that are easier to accelerate at the expense of these offering more modest compression ratios.

Research Question: *Can we design a compression framework that upon deployment decompresses the model parameters on-the-fly in a lightweight manner and, achieves inference acceleration, not by doing less compute, but by reducing data movement?*

3.1 Problem Setting and Contributions

This chapter presents a new paradigm for compression frameworks aiming at accelerating inference by reducing the impact of data movement. Concretely, it address the movement costs associated to the fetching the model parameters and their transfer across the memory subsystem during inference. Under this formulation, the main challenge is to design a compression strategy that, not only compress the model while maintaining application-specific performance metrics (e.g. accuracy), but that also can be reverted efficiently, without outweighing the savings due to data movement.

We address this dual challenge by introducing the use of orthogonal variable spreading factor (OVSF) codes to the ML domain and, treat them as a base for the high-dimensional space where model parameters are defined. As it will be presented in Section 3.3.1 in greater detail, these codes are binary, deterministic and can be generated on-the-fly very cheaply. These properties enabled the design of compact models for applications as diverse as keyword spotting or image classification, with model sizes ranging from 14K to 26M parameters, while maintaining negligible accuracy drops in most cases. We study several ways of generating the model parameters using OVSF codes and, derive and implement a hardware weights generator capable of sustaining high processing rates without slowing down layer inference, even in setups with constrained memory bandwidth. This weights generator co-resides with an FPGA-based single computation engine in charged of performing the convolution between the generated weights and the layer input. The framework presented in this chapter, which we name `unzipFPGA`, achieves up to $1.7\times$ lower latency on FPGAs with restricted memory bandwidth compared to state-of-the-art structured-pruning approaches.

The remaining of this chapter is organized as follows: in Section 3.2 we review the set of approaches found in the literature that contributed to the design of compact model architectures, then we present a framing for *on-the-fly* models and, compare it to existing frameworks with similar characteristics. The section ends with an overview of FPGA-based

CNN inference engines and their limitations. In Section 3.3, we introduce how on-the-fly models can be constructed using OVSF binary codes and, present a hardware weights generator suitable for single computation engines. Following a detailed description of the experimental setup in Section 3.4, an extensive evaluation of the compression and acceleration capabilities of on-the-fly OVSF models is presented in Section 3.5. This chapter concludes with a discussion section, where avenues for future work are presented.

3.2 Background and Related Work

The field of neural network compression attracted exceptional levels of attention soon after the first CNN-based architectures, e.g. AlexNet [200] and VGG [305], became the reference designs for future architectures, making prior supervised learning attempts relying on hand-crafted features obsolete [142, 82, 83, 81]. The development of compression frameworks took place parallel to other attempts focusing on reducing both memory footprint and compute costs of large, over-parameterized models, with lightweight architecture designs (e.g. MobileNets [165]) swiftly becoming the preferred option. However, with the rapid advancements in NAS [420, 42, 283, 318, 230, 47] techniques in the following years and, training techniques enabling the deployment of accurate low-precision arithmetic models [175, 198, 104, 309, 242], the interest for *compression*-only frameworks stagnated.

In this section we provide an overview on popular compression frameworks proposed in recent years. While the main objective of such frameworks is to reduce the model size, they occasionally include methods (e.g. weight pruning, as first introduced in Section 2.1.1) that can also translate into acceleration of inference stages. In the following pages, we first introduce some of these frameworks applied to CNN-based image classification tasks. Then, we introduce *on-the-fly* models and state how these differentiate from other compression frameworks. Finally, we provide an overview on single computation engines for CNN workloads which will be relevant when introducing the proposed `unzipFPGA` framework.

3.2.1 Compression Frameworks for CNNs

Over the years, multiple compression methods aimed at reducing the memory footprint of CNN models have been proposed. These can be categorized into:

Low-rank approximations. Early forms of compressing parameters of convolutional and fully connected layers relied on well established matrix factorization and dimensionality reduction methods. Techniques such as truncated single value decomposition (SVD) [77,

412, 383, 205, 382] and, to a lesser extent, principal component analysis (PCA) [112, 41, 229], became popular choices to obtain compress representations of the weights matrices. These, however, soon became outperformed in terms of both compression ratio and accuracy retention by data-driven approaches such as pruning and quantization that better exploited the plasticity of CNNs.

Pruning. By discarding parts of the model that are redundant, smaller lightweight model architectures become more suitable for deployment on constrained settings with limited memory and compute budgets. Han et al. [141] reintroduced the use of pruning on modern CNNs, after several attempts were first made in the early 90s [261, 177, 143] for much smaller models, and achieved $13\times$ compression on the then state-of-the-art VGG network with no accuracy drop. Since then, the sophistication of pruning methodologies has grown rapidly: pruning by measuring the level of sparsity on the output feature maps [167] or their correlation[252]; by discarding channels with minimal contribution to the loss [257]; or, by estimating which elements to prune before training [343, 212, 325], enabling the training of smaller model directly. In Section 2.1.1 we provided further details on how pruning works and, highlight variations of this optimization technique found in the literature.

Clustering and Code-book approaches. Weight clustering, often via K-means, has been used as an effective weight sharing technique [173, 77, 373]: in DeepCompression [140], weights with similar values are clustered together and a codebook is generated; similarly but in the Fourier domain, CNNPack [354] clusters nearby DCT frequency coefficients; in Deep k-means [373], a regularization mechanism is used to learn filters where rows below to one of the k-rows clusters. The centroids and clustering indices generated by these methods can be further compressed by either quantizing the centroids and re-calibrating their values, by means of lossless compression techniques (e.g. Huffman encoding [119]) [113, 66], or both, as is the case in DeepCompression and CNNPack.

Quantization. Quantization allows for model size reduction and inference speedup without changing the model architecture. The most extreme form of model quantization is binary networks [72, 228, 376], which replace convolutions with bit-shifts resulting in $58\times$ inference speed-ups [281]. Ternary and 2-bit models [215, 341, 123] achieve higher accuracies while alleviating some the challenges of training binary networks [11]. However, it is 8-bit quantization [175, 198, 348] that has attracted the most attention due to its balance between accuracy, model size reduction and inference speedup. Whether it is applied after training [372, 140] or during training [175, 309, 347], quantization has become, along with pruning, the go-to methods for reducing memory and computational footprints of CNNs.

We refer the reader to Section 2.1.2 for further details on how quantization works and, how it has been used to compress models and accelerate inference in the recent literature.

Knowledge distillation. When a large and accurate model is available, its generalisation capability and knowledge about the task at hand can be transferred to a smaller network [157]. In the literature, the former is often referred as to *teacher*, while the latter gets labeled as *student* network. Knowledge distillation (KD) was first framed as a logits transferring problem by which the student network uses the soft labels outputted by the teacher as targets for guiding the training process [157, 344, 220, 61]. However, logits transfer alone can result in the student network to fail when generalizing to unseen inputs. To alleviate this, a *teacher assistant* module can be introduced to guide the mapping of features between teacher and student, preventing overfitting at the expense of increasing the complexity of the training process [251, 267, 109]. In certain settings, the teacher and student would be using different (albeit related) datasets. For example, the former could have be trained on ImageNet while the latter has to generalize to a smaller proprietary dataset for image classification. For KD to work, several domain adaptation techniques have been proposed [93, 390] and introduced during training, often as regularization terms added to the loss. We refer the reader to the survey work of Gou et al. [129] for a detailed overview on existing knowledge distillation methods.

While most of the works previously referenced focus on a single compression strategy, others make use of several of these techniques and construct in this way multi-stage compression frameworks: the already mentioned DeepCompression stacks pruning, quantization, clustering and lossless encoding stages; CNNPack uses the same set of techniques excluding the initial pruning phase; or Adadeep [231], which relies on SVD, magnitude-based weight pruning and separable filters as in Bhattacharya & Lane [35].

3.2.2 On-the-fly Compression Frameworks

The techniques previously mentioned successfully construct compact model representations offering different levels of trade-offs between compression ratio and application performance (e.g. classification accuracy) However, they are susceptible of requiring a symmetric decompression stage before inference can begin. For example, if a framework used unstructured pruning and stored the sparse weights in CSR format, but the software stack on the target device does not support sparse operations (or they come with significant overheads), transforming the weights into their dense representation would be needed, increasing the runtime memory as a result. Similarly, if a model has been compressed using heterogeneous bitwidth quantization (e.g. 8-bits and lower) but the target platform does

not natively support such numerical resolutions, casting to the closest supported bitwdith would be required, likely resulting in higher memory footprint. When these *hidden*[1] costs arise, the benefits of having a compressed model on-device are voided the moment the model performs inference and, as a result, the compression benefits get relegated to easing the impact of distributing such models over the network [282] or when storing them.

We define *on-the-fly* compression methods as those that explicitly account for such decompression stage on the target device and include their associated compute and memory overheads at design time. On-the-fly methods trade model size and data movement costs for additional compute when decompressing each layer at run time. The challenge is to design a compression strategy that significantly reduces model size, preserves model accuracy and, is computationally simple to revert. Concretely we label as *on-the-fly* those compression-first frameworks that: (1) require a single-step *inflation* stage by which the compressed model parameters are fully materialized; (2) such inflation stage is performed on-demand, i.e. at each layer and for each input, without requiring the entire model to be fully decompressed on-device; therefore, (3) the inflation procedure has to be lightweight and without requiring such processing to be amortized over many inference stages; finally, (4) on-the-fly methods use compression as a means to reduce data-movement overheads, balancing the costs of decompressing the model parameters with the latency savings due to reduced off-chip or main memory accesses.

From the literature, we can identify several compression methods that, although never framed in such way by the authors, could be labeled as on-the-fly. For example, compression frameworks relying on truncated SVD representations of fully-connected or convolutional layers [77, 412, 383, 205], where parameters can be constructed by reverting the matrix factorization stage at run time. In HyperNetworks [136], an auxiliary neural network is used as a universal generator for each layer's weights in the main network given a low-dimensional embedding. Both, the weights generator and the layer embeddings are trained end to end, with the former being shared across layers or groups of layers. We see HyperNetworks as the prototypical on-the-fly compression framework. However, since their introduction and due to the memory overheads of the generator network, the use of hypernetworks has shifted instead to methods that focus on their adaptive generation capabilities [42, 233, 311, 339], instead of on their compression properties. In DCFNet [277], filters are constructed as a dense combination of Fourier-Bessel bases that are generated deterministically at run time. Since the basis are deterministic, a negligible amount of index-

[1] We call them *hidden* because they only become evident after deployment on specific platforms and, when the model needs to be used for inference. With an specialized accelerator [139], multi-stage compression frameworks such as DeepCompression can be accelerated.

ing parameters are needed for their construction, reducing the model size to just the weighting coefficients. These properties made this method the closest to the work presented in this chapter. In addition to Fourier-Bessel bases, the authors also evaluate the compression capabilities of using random bases and, PCA bases computed from a pre-trained model. DCFNet can also be implemented in such a way that the explicit generation of filters is avoided. This is achieved by first convolving the inputs with the basis and then performing a linear combination of the intermediate outputs. Although such formulation translates into latency savings, this DCFNet variant cannot be considered as on-the-fly. Finally, in WSNet [184] training a CNN is framed as learning a single weight matrix per layer. Then, the filter tensor in a given layer is constructed by a tiled sampling of the weight matrix, inducing in this way high levels of parameter sharing across channels. The sampling is deterministic and defined at training time. If filters are constructed from overlapped patches, this will result in redundant computations during inference. This can be avoided by framing such convolution as variant of the integral image algorithm [337], which speedups the inner product at the expense of some pre-processing overheads for both operands. WSNet was recently extended [393] adding better support for $3{\times}3$ convolutions.

3.2.3 FPGA-based CNN Inference Engines

The emergence of CNNs as a core component of modern ML systems spawned an interest for custom FPGA-based accelerator designs during the last half of the decade. This trend continues today. Among the plethora of designs, one of the most widely adopted paradigm is the single computation engine [132, 4, 197, 408, 379, 196], where a powerful processing engine is time-shared to sequentially execute the layers in the CNN. This approach enables the reuse of the accelerator resources across various CNNs and minimizes the need for fabric reconfiguration upon deployment. In other words, they offer a balanced trade-off between programmability and performance.

3.2.3.1 Single Computation Engines for tiled GEMM-based Convolutions

To execute various types of layers with operands of varying shapes, the core of a CNN engine comprises a parametrized array of processing elements (PEs) that execute a block general matrix multiply or GEMM. By interpreting convolutional and linear layers as a matrix multiplication, both types of layers can be described under the same formulation. Borrowing notation from Kouris et al. [196], convolutional layers can be described as:

$$\text{CONV} < H, W, N_{in}, N_{out}, K_H, K_W, S_H, S_W, Z > \tag{3.1}$$

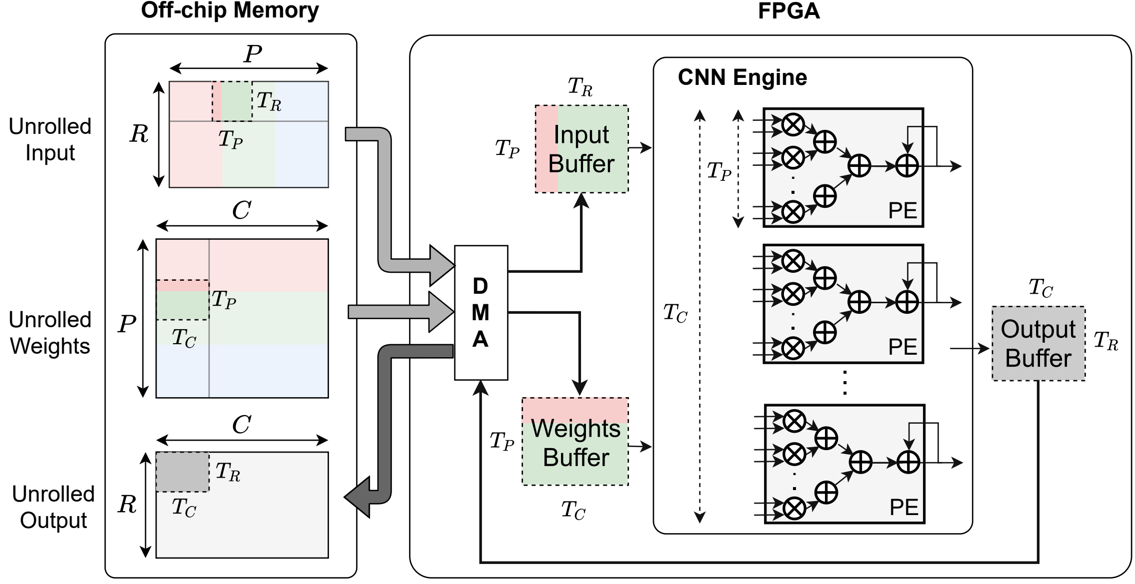


Figure 3.1: Overview of the main components in a typical GEMM-based CNN FPGA engine. Unrolled in memory input of dimensions $1{\times}N_{in}{\times}H{\times}W$ for a convolutional layer with $K{\times}K$ and N_{out} output channels are loaded via Direct Memory Access (DMA) on to the respective FPGA buffers. This process is done in a $\langle T_R, T_P, T_C\rangle$ tiled fashion, generating a single $T_R{\times}T_C$ output tile after accumulating over $\left\lceil \frac{P}{T_P} \right\rceil$ matrix multiplications.

where H and W denote the height and width of the input tensor, N_{in} and N_{out} the number of input and output channels, K_H and K_W the spatial dimensions of the filters, S_H and S_K the striding applied along each dimension and, Z the zero padding for the inputs. Similarly, the dot products in linear layers can be formulated as:

$$\text{CONV} < 1, 1, N_{in}, N_{out}, 1, 1, 1, 1, 0 > \tag{3.2}$$

In this manner, a CONV layer with N_{in} $H{\times}W$ input activations, N_{out} output channels, $K{\times}K$ filters, Z zero padding and S stride involves the multiplication between an $R{\times}P$ unrolled activations matrix and a $P{\times}C$ unrolled weights matrix in order to produce a $R{\times}C$ output matrix, with $R{=}\left\lceil \frac{H+2Z-K}{S} + 1 \right\rceil \left\lceil \frac{W+2Z-K}{S} + 1 \right\rceil$, $P{=}N_{\text{in}}K^2$ and $C{=}N_{\text{out}}$. In Figure 3.1 we illustrate the main components involved in the design of an FPGA-based CNN engine using the syntax presented in this subsection.

The GEMM unit is parameterized by the triplet $\langle T_R, T_P, T_C\rangle$, which results in tiling the multiplication of the unrolled matrices along each dimension $\langle R, P, C\rangle$. The choice of tiling dimensions is generally a function of the number available PEs (T_C) and their width (T_P). In this way, $\left\lceil \frac{P}{T_P} \right\rceil$ $T_R{\times}T_P$ (input) and $T_P{\times}T_C$ (weights) tiles are loaded from off-chip memory, multiplied and, accumulated over producing a $T_R{\times}T_C$ output tile. This is equivalent to an output stationary dataflow [60, 89]. In this chapter we follow this dataflow

since its design minimizes the energy consumption related to the reading and writing of partial computations during the GEMM, alleviating in this way the data movement costs and, aligning with the objectives of on-the-fly formulation previously introduced.

3.2.3.2 Limitations of single computation CNN engines

Despite the flexibility introduced with single computation designs, performance is often bounded by two factors: layers with low compute-to-communication ratio that become memory-bound [196, 266, 239]; and, suboptimal mapping of diverse CNN layers on the fixed engine configuration, leading to underutilized PEs [239, 332, 407] due to the mismatch of diverse layer shapes (e.g. 3×3 vs 1×1 convolutions). These two factors set a hard limit to the actual sustained performance that this family of accelerators can reach, indicating an emerging need for novel solutions to minimize their impact. These limitations are exacerbated as multiple applications are collocated on a single device. The work presented in this chapter focuses on alleviating the memory-boundness of CNN layers by storing their compressed weights representation on-chip or, in main memory for larger models, and always reconstruct them on-the-fly. Intuitively, this should also translate into a better utilization of the available computational resources.

The memory bandwidth problem faced by CNN engines has been studied in previous work. EIE [139] uses the multi-stage DeepCompression framework to significantly compress (over 95% compression ratio) parameters in fully connected layers. However, as these layers have been mostly abandoned in modern CNNs, its applicability is limited. Angel-Eye [134] compresses all layers through precision quantization. Cambricon-X [410] transfers only the non-zero weights, while Cambricon-S [417] and Scalpel [399] apply coarse weight pruning, but with significant accuracy drop. Shen et al. [297] exploits large batch sizes to increase weights reuse and, thus, is not suitable for latency-critical applications that cannot tolerate batching [332]. Focusing on reducing the memory footprint of layer activations, Chen et al. [13] fuses adjacent layers to cache intermediate activations, while Eyeriss [60] and others [258, 270] employ encoding schemes to minimize their bandwidth footprint. Other solutions have either relied on large devices [22] and multiple FPGAs [106] to fit all weights on-chip, or utilized highly customized designs to exploit multi-precision cascades [197] or fine-grained pruning [238] at the cost of notable accuracy drop.

3.3 On-the-fly Weights with OVSF Codes

In this section we identify Orthogonal Variable Spreading Factor (OVSF) codes as an attractive choice to facilitate the generation of model parameters on-the-fly. We show that these can be used as a base over the real field, guaranteeing models using them to be indistinguishable from regular models that are not on-the-fly. These codes can be constructed recursively following a deterministic algorithm, largely simplifying the design of the code generator instance running along side the network inference process. The deployment of such models could be simplified even further by not generating the OVSF codes on demand but storing them instead as part of the model. Since these are binary and, shared across channels and layers, they represent a negligible portion of the artifact to deploy.

This section first provides a brief historical context for OVSF codes and two approaches to construct them. Then, we show how OVSF codes are used to construct filters in CNNs, how they are introduced in the training process and, limitations that arise when using them. We then introduce an adaptive regression stage that enables transforming standard models into on-the-fly models requiring just a few iterations of re-training. Finally, we describe `unzipFPGA`, a framework that introduces a novel CNN hardware architecture with custom memory organization and datapath for on-the-fly generation of CNN weights in single computation FPGA-based CNN engines.

3.3.1 Re-purposing OVSF Codes for Model Compression

The chosen OVSF codes are a set of mutually orthogonal binary codes originally designed to split in the frequency domain signals from different users in W-CDMA based 3G cellular systems [8]. Using them as channelization codes allowed for communication channels remain orthogonal in multi-user access scenarios, reducing signal interference while dramatically increasing system capacity.

These codes can be obtained using Sylvester's construction algorithm for Hadamard matrices. In this way, given $H_0 = [1]$ and H_2, subsequent H_{2^k} expansions are defined by:

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad H_{2^k} = \begin{bmatrix} H_{2^{k-1}} & H_{2^{k-1}} \\ H_{2^{k-1}} & -H_{2^{k-1}} \end{bmatrix} = H_2 \otimes H_{2^{k-1}} \tag{3.3}$$

where H_{2^k} is an $L \times L$ Hadamard matrix, with $L = 2^k, k \in \mathbb{N}$ and $\otimes$ is the Kronecker product. Each row for $k > 1$ is an OVSF code fulfilling the properties of being binary and orthogonal to each other. This will enable us to use them as basis for $\mathbb{R}^L$ when defining the filter construction process in Section 3.3.1.1. An alternative formulation [98], allows for

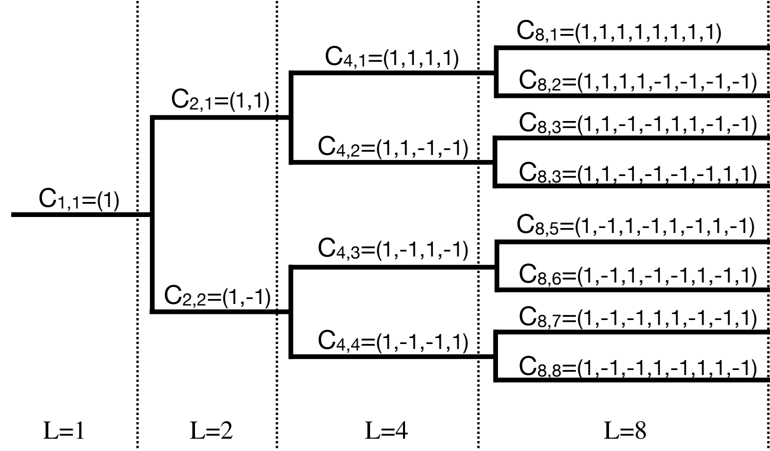


Figure 3.2: Fully constructed binary tree with OVSF codes of lengths $L \in 2, 4, 8$. Each code can be individually addressed as $C_{L,K}$ where L is referred to as *spreading factor* in the telecoms jargon and K the *code index*. For example, $C_{2,2}$ is equal to the second row in H_2 in Equation (3.3).

the construction of OVSF codes as a recursive expansion of a perfect binary tree. Such tree is depicted in Figure 3.2 for OVSF codes of up to length $L = 8$. This formulation allows the retrieval of individual codes, at both leafs and intermediate nodes, without having to explicitly generate the entire tree. This property of lightweight generation of individual OVSF as well as the fact that the construction process is recursive and deterministic, make OVSF codes an attractive basis over which ML models can be defined and generated on-the-fly. In addition and, unsurprisingly given the maturity of the W-CDMA standard, several efficient hardware implementations of OVSF code generators have proposed [15, 190, 286, 275] by the wireless community.

3.3.1.1 Constructing filters with OVSF Codes

By treating the set of L OVSF codes as a base spanning $\mathbb{R}^L$, we can define the construction process of an arbitrary real-valued vector $\boldsymbol{v}_i'$ as the linear combination of such codes:

$$\boldsymbol{v}_i' = \sum_{j=0}^{\lfloor \rho \cdot L \rceil} \alpha_i^j \boldsymbol{b}_i^j, \qquad E_i = \|\boldsymbol{v}_i' - \boldsymbol{v}_i\|_2^2 = \left\| \sum_{j=0}^{\lfloor \rho \cdot L \rceil} \alpha_i^j \boldsymbol{b}_i^j - \boldsymbol{v}_i \right\|_2^2 < \epsilon \tag{3.4}$$

where $\boldsymbol{\alpha}_i = \{\alpha_i^0, \alpha_i^1, \alpha_i^2, ..., \alpha_i^{L-1}\}$ are weighting coefficient, $\boldsymbol{b}_i^j$ is the j-th OVSF binary code of length L and, $\rho \in [\frac{1}{L}, 1]$ is the ratio of codes to use in order to construct $\boldsymbol{v_i}$. The expression on the right measures the difference between a real-valued standard vector of length L, $\boldsymbol{v}_i$, and $\boldsymbol{v}_i'$. Intuitively, $\epsilon \to 0$ as we increase the ratio of binary codes used.

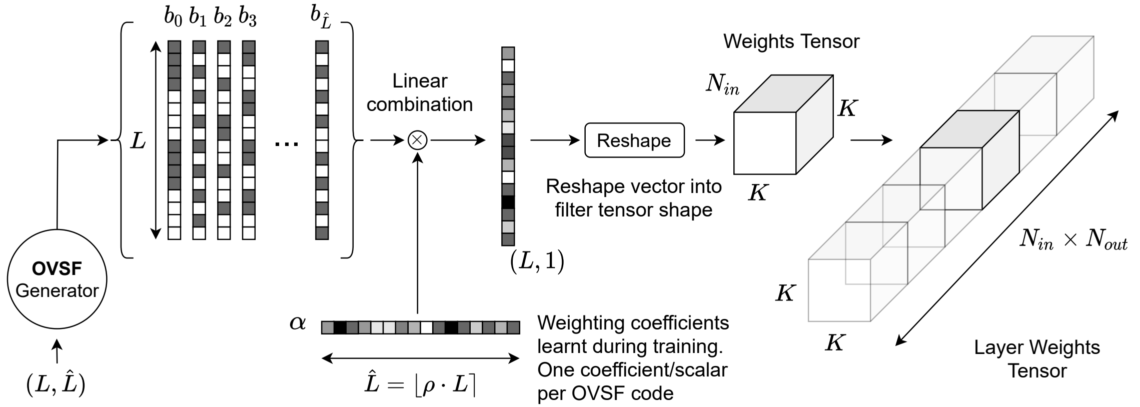


Figure 3.3: Constructing filters of a CNN layer using OVSF codes. A filter of shape $N_{in} \times K \times K$ is obtained by performing a linear combination of $\hat{L} = \lfloor \rho \cdot L \rceil$ of length $L = N_{in} \times K \times K$, with $\rho \in [0, 1]$. Then the $N \times 1$ vector is reshaped to the target filter's shape. If the convolutional layer has N_{out} output channels, this process is repeated that many times concatenating the results.

When constructing matrices from OVSF codes or higher-dimensional tensors, a reshaping stage follows the linear combination shown in Equation (3.4). In this way, if the weights tensor of a given convolutional layer is of shape $N_{out} \times N_{in} \times K \times K$, the construction process using OVSF could be framed as the concatenation of N_{out} $N_{in} \times K \times K$ filters using codes of length $L = N_{in} \times K \times K$ and up to L of such codes. This scenario is illustrated in Figure 3.3. This process can be constructed at different granularity levels, as discussed later in Section 3.3.2, each with a different compute overhead.

In certain scenarios, a pre-trained model with standard convolutions might be available. In such cases, the formulation in Equation (3.4) could be reinterpreted as a minimization problem and regress the set of $\boldsymbol{\alpha}_i^*$ that minimize the difference w.r.t the standard filter $\hat{f}_i$ as $\boldsymbol{\alpha}^* = \text{argmin}_\alpha \left\| f - \hat{f} \right\|_2^2$, which can be implemented as a 2-layer MLP regression stage. We provide further details on how this is implemented in Section 3.4 with results in Section 3.5.

3.3.2 On-the-fly OVSF Models: Training and Limitations

Unlike standard CNNs, architectures using OVSF codes do not learn convolutional filters directly. Instead, they learn weighting coefficients for each OVSF code. In the forward pass, the filters are individually generated prior to convolving with the input. Then, inference proceeds as normal. During back-propagation, only the weighting coefficients $\{\alpha\}_{k=1}^{L}$ of each layer are updated.

Despite the simplicity of OVSF codes as a straight drop and replacement option for standard convolutional layers, the nature of OVSF codes and the filter generation process, presents several challenges:

- **OVSF codes are of power-of-two length.** This constrains the generation of filters with all N_{out}, N_{in}, and K being a power-of-two integer. While this might be reasonable for the input and output channel dimensions, it prevents the construction of $3{\times}3$ filters, which are now ubiquitous in modern architectures. We evaluated two approaches to overcome the limitation of generating $3{\times}3$ filters using OVSF codes: first, by learning a $3{\times}3$ crop from a $4{\times}4$ filter (which can be represented using OVSF codes); secondly, by means of an auxiliary layer that performs adaptive average pooling transforming a $4{\times}4$ filter into the desired $3{\times}3$ shape.

- **Choice of basis.** Model compression is only achieved when $\rho < 1$, which raises the question of which bases to choose from the total L available for OVSF codes of length L. One approach would involve choosing the first $\hat{L} = \lfloor \rho \cdot L \rceil$, potentially allowing for further optimizations on the OVSF code generator design. Alternatively, basis could be greedily dropped during training based on the magnitude of their associated weighting coefficient. A similar strategy has been adopted by some pruning frameworks. We analyze the impact on model accuracy for these two strategies for choosing OVSF codes that span $\mathbb{R}^L$.

- **Filter generation overheads.** Despite the simplicity of the filter construction process, it can account for a sizeable portion of the compute needed for inference at each layer. The procedure presented earlier and described in Figure 3.3 is of complexity $\mathcal{O}(N_{out}(N_{in}KK)^{\frac{\hat{L}}{L}+1})$, which becomes a problem at deeper layers of the network. The construction process can be simplified by framing it as a 2-level concatenation where first N_{in} $K{\times}K$ OVSF-based matrices are constructed and concatenated. Then this process is repeated N_{out} times, reducing the computational complexity to $\mathcal{O}(N_{out}N_{in}(KK)^{\frac{\hat{L}}{L}+1})$. Both approaches share the same properties, including the same number of parameters when perfectly (i.e. $\rho = 1$) constructing an arbitrary filter, but at different granularities.

3.3.3 A Hardware Weights Generator using OVSF Codes

Of particular concern when it comes to the deployment of models using on-the-fly weights constructed from OVSF codes are the costs associated to generating the filters at deeper layers in the network. As these costs grows at least linearly with the number of channels, in order to attain high performance, the weights generation algorithm is mapped to hardware via two techniques: a *tiled weights generation* (TiWGen) stage and a hardware weights generator (`CNN-WGen`) block. Together, they represent the hardware-level contribution of the proposed `unzipFPGA` framework. Their parameterization is exposed to the

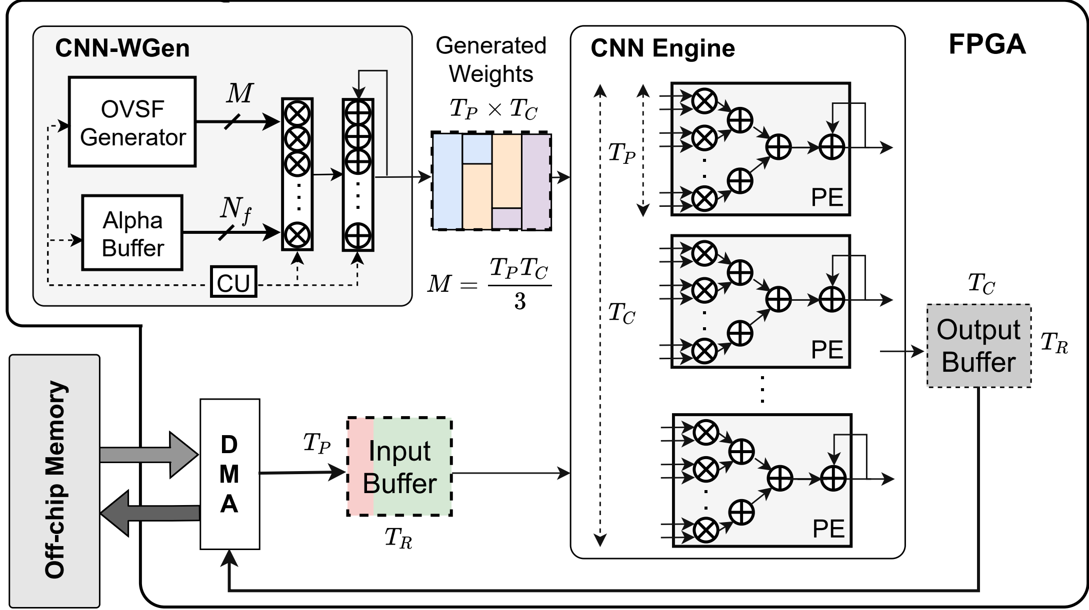


Figure 3.4: Overview of `unzipFPGA`'s architecture. In this example, the weights tile is split into three sub-tiles that are generated sequentially by `CNN-WGen`. The OVSF Generator uses a FIFO buffer to store OSVF codes and, logic to serve and re-insert them efficiently. The Alpha Buffer stores the $\{\alpha\}_{k=1}^{L}$ coefficients learnt during training. The combination of OVSF codes is performed in the M-wide array of multipliers and adders. Then, the corresponding sub-tile in the $T_P \times T_C$ tile is updated. The control unit (CU) resets the state of the accumulators between sub-tiles updates.

system-level design space exploration (in A.4) to yield the highest performing allocation of resources between `CNN-WGen` and the CNN engine for a given CNN-FPGA pair.

As introduced in Section 3.2.3 and illustrated in Figure 3.1, convolutions and fully connected layers can be interpreted as block-wise matrix multiplications, where tiles in the unrolled input and weight matrices are multiplied in the CNN engine comprised of an array of PEs. Under the on-the-fly formulation previously presented, the layer weights are not present in off-chip memory as these have to first be generated by linearly combining $\{\alpha\}_{k=1}^{L}$ coefficients with their respective OVSF codes. The $T_P \times T_C$ weights tile is generated by `CNN-WGen`. Figure 3.4 shows the complete architecture used for `unzipFPGA`. The remainder of this section describes its implementation and design choices.

3.3.3.1 Tiled Weights Generation

To be able to target layers of various dimensions, we introduce a sub-tiling method that divides each $T_P \times T_C$ weight tile into $\lceil \frac{T_P T_C}{M} \rceil$ sub-tiles of length M. In this way the data flow of diverse layers are identical to each other. Furthermore, M provides a tunable trade-off between weights generation speed and resource consumption: higher values of M would

Algorithm 1: Generation of a layer's weights using TiWGen. Each $T_P \times T_C$ tile of the weights matrix is processed sequentially (line 1) by partitioning each tile into $\left\lceil \frac{T_P T_C}{M} \right\rceil$ sub-tiles (line 2). After all basis vectors of the current sub-tile have been processed (lines 4-9), the associated part of the output tile is updated, proceeding then to the next sub-tile. When all sub-tiles of a tile have been generated, the weights matrix is updated (line 12) and the algorithm continues to the next tile.

Input: Layer's weights matrix shape: $P \times C = N_{\text{in}} K^2 \times N_{\text{out}}$, row and column tile sizes T_P and T_C, α values with $\alpha \in \mathbb{R}^{N_{\text{in}} N_{\text{out}} \lceil \rho K^2 \rceil}$

Output: Weights matrix $\boldsymbol{W}$

```
1  for t ← 1 to ⌈P/T_P⌉·⌈C/T_C⌉ do // tiles loop - # PIPELINE
2      for i ← 1 to ⌈T_P T_C / M⌉ do // sub-tiles loop - # PIPELINE
3          sub-tile_i^t ← 0
4          for j ← 1 to ρK^2 do // basis vectors loop - # PIPELINE
5              for k ← 1 to M do // # UNROLL
6                  incr_k ← vec_j(k) · α_k // Multiplier array
7                  sub-tile_i^t(k) ← sub-tile_i^t(k) + incr_k // Adder array
8              end
9          end
10         tile^t ← UpdateTile(tile^t, sub-tile_i^t)
11     end
12     W ← UpdateMatrix(W, tile^t)
13 end
```

result in fewer but larger sub-tiles requiring more compute resources to instantiate the required M-wide vector units in CNN-WGen; on the other hand, for lower values of M, fewer resources would be needed at the expense of requiring more steps to fully generate the weights tile since, as described in Algorithm 1, this is a sequential process.

3.3.3.2 A Hardware Weights Generator

The microarchitectural design of CNN-WGen is comprised of three main components: a compute datapath formed by two vector arithmetic units (multiplier and adder arrays); the Alpha buffer storing the α values; and, the OVSF generator responsible for outputting OVSF codes to generate the M-sized vector sub-tiles as dictated by the TiWGen scheme.

CNN-WGen's strategy for mapping TiWGen pipelines the three outer loops over tiles, sub-tiles and basis vectors, and unrolls the inner loop that processes the M-sized sub-tile. For such unrolling, CNN-WGen employs two M-wide vector units that perform M-parallel multiplications and additions, respectively. In this manner, tuning M can balance the parallelism-resource usage trade-off.

Compute Datapath. The vector arithmetic units in the `CNN-WGen` block must have a fixed number of inputs that meets the DSP resources of the target FPGA. For the i-th sub-tile (line 3 in Alg. 1), ρK^2 vectors of size M are produced by the OVSF generator and, the associated α values fetched from the Alpha buffer, are fed to the multiplier array in a pipelined manner. All M elements are processed in parallel by the M-wide vector units, leading to the unrolling of the inner loop on line 5 of Alg. 1. The adder array processes the outputs of the multiplier array by accumulating the ρK^2 intermediate results. Finally, when the processing moves to the next sub-tile, the control unit (CU in Fig. 3.4) resets the accumulators' state. The notation using K accounts for weights already being adjusted to be 3×3 or other $K \in \mathbb{Z}$, not restricted to a power-of-two number.

Alpha Buffer. Following TiWGen, each sub-tile contains weights from N_f distinct $K\times K$ filters as in Eq. 3.5. To sustain the throughput of `CNN-WGen`, an equal number of αs has to be fetched in parallel from the Alpha buffer. This is accomplished by designing the buffer with appropriate memory organization and addressing. Each layer contains $N_{\text{in}}N_{\text{out}}\lceil \rho_l K_l^2 \rceil$ distinct α values. The RAM blocks are organized with $N_P^{\text{Alpha}}{=}N_f$ ports and a depth of D^{Alpha} as in Eq. 3.6 to accommodate N_L layers.

$$N_f = \left\lceil \frac{\min(T_P, M)}{K_{\max}^2} \right\rceil \left\lfloor \frac{M}{T_P} \right\rfloor + \text{mod}(M, T_P) \left\lceil \frac{M}{K_{\max}^2} \right\rceil \tag{3.5}$$

$$D^{\text{Alpha}} = \overbrace{\sum_{l=1}^{N_L}}^{\text{for each layer}} \frac{\overbrace{N_{\text{in}}^l N_{\text{out}}^l \lceil \rho_l K_l^2 \rceil}^{\text{no. of } \alpha \text{ values}}}{N_P^{\text{Alpha}}} \quad (\textit{ Buffer depth }) \tag{3.6}$$

where N_L is the number of layers, $N^l_{\{\text{in,out}\}}$ the l-th layer's number of input/output channels and ρ_l the compression ratio.

OVSF Generator. The generation of OVSF codes is also done in a blocked manner with a tile size of M. To produce the i-th sub-tile, the OVSF generator has to feed the compute datapath with ρK^2 codes that are tiled as dictated by TiWGen's parameter M. In order not to slow down the operation of `CNN-WGen`, the OVSF generator has to match the processing rate of the vector units by providing a bandwidth of M bits/cycle. To achieve this, the OVSF generator utilizes an OVSF FIFO buffer storing the $(K_i^2 K_i^2)$-bit OVSF codes for a given layer i. Since these are binary, they can be stored in a very compact manner. Further details and a diagram af the OVSF generator are presented in Appendix A.2.

3.4 Experimental Setup

On-the-fly models with OVSF codes are evaluated along two dimensions: first, we focus on measuring their performance in terms of model accuracy at different compression ratios for both image and audio classification tasks; then, we evaluate the attainable increase in throughput that results when generating the layer weights on-the-fly with `CNN-WGen` under the `unzipFPGA` formulation as opposed to retrieving them directly from off-chip memory. In this section, we describe the experimental setup for both set of experiments and provide details about the model architectures, the datasets and, the FPGA platforms used.

3.4.1 Datasets and Tasks

We evaluate on-the-fly models on image classification and speech classification tasks. The former is evaluated on ResNet-{18,34,50} [146] and SqueezeNet 1.1 [173] on CIFAR-10 and ImageNet. For the latter we implement a custom 3-layer CNN architecture that consumes MFCC[2] representations of the inputted audio samples from the SpeechCommands [358] dataset. Datasets and architectures, as well as their parameterization, are described next.

Image classification. We conducted our experiments on two popular datasets for image classification: ImageNet and CIFAR-10. ImageNet is a large-scale image classification dataset, which contains 1000 categories and a total of 1.33 million color images. These images vary in dimension and resolution and are generally resized and cropped to 224×224 images. The dataset is divided into training and validation data, with 1.28 million images and 50,000 images, respectively. The CIFAR-10 dataset contains 60,000 32×32 color images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images, with equal number of images per class in both training set and test set.

We have implemented OVSF-based convolutional layers and their associated training pipeline on top of PyTorch (1.5) [272]. To derive the OVSF models, we modified the official PyTorch-based ResNet architectures by replacing all 3×3 convolutional layers within residual blocks with their OVSF counterparts. In all our ImageNet experiments, we employed pretrained models from *torchvision* (0.6.0) and transformed them into their OVSF counterparts following a regression stage. Then, the models were fine-tuned for 30 epochs using Adam [191] and learning rate decay every 10 epochs. For each given model, we trained two OVSF variants following different distributions of ratios ρ for each 3×3 convolutional

[2]Mel-frequency cepstral coefficients (MFCCs) are commonly used as features in speech recognition systems. These are hand-crafted features and part of the ETSI [73] standard for mobile communication systems.

layer in each of the four residual blocks. First, OVSF50 with ratios=$[1.0, 0.5, 0.5, 0.5]$; and OVSF25 with ratios=$[1.0, 0.4, 0.25, 0.125]$. We follow the same procedure and ratios for SqueezeNet's *Fire* modules. For CIFAR-10, we train all models from scratch using the same architectures as for ImageNet with the exception of having smaller $3{\times}3$ input convolutional layers, no pooling layer after the first convolutional layer and, reduced final fully connected layer outputing a 10-element tensor. We trained models for 180 epochs using batch size of 128 and, initial learning rate of 0.1. In addition to OVSF50 and OVSF25, we evaluate the impact on model performance for: OVSF100, which keeps $\rho{=}1$ across all layers; OVSF75, with ratios=$[1.0, 0.5, 0.6, 0.6]$ which results in model sizes close to the baseline models using $3{\times}3$ filters; and, OVSF12.5 with ratios=$[0.75, 0.25, 0.125, 0.125]$ which compresses these further than OVSF25. Table A.3 includes a detailed breakdown of the ρ assigned to each layer for the architectures considered in this chapter.

While the main aspect here evaluated is the accuracy retention at different compression ratios, we also evaluate: two different approaches to extract $3{\times}3$ filters from $4{\times}4$ since the former cannot be constructed with OVSF codes, as first discussed in Section 3.3.2. We also compare the results between choosing the first $\lfloor \rho \cdot L \rceil$ basis when $\rho{\leq}1$ or greedily selecting them during training.

Audio Keyword Spotting (KWS). KWS has become a popular always-on feature in smartphones, wearables and smart home devices. It serves as the entry point for speech based applications once a predefined command (e.g. "Ok Google", "Hey Siri", "Alexa") is detected from a continuous stream of audio. Because KWS applications are always running they follow a very efficient architectural design and are often implemented on small dedicated microcontrollers (MCUs). These devices are constrained in terms of memory and compute capabilities, limiting the complexity and memory footprint of the deployed model. We used this task to evaluate the performance of on-the-fly models with OVSF codes when scaling them down to models of memory footprint <25 KB.

We employ Google's Speech Commands dataset, which is comprised of 65k one-second long single-word audio clips from thousands of different people. There are 30 different keywords that can be classified into three groups: a known expression, commands like "yes", "no" or "up", "down"; silence (i.e. a audio clip with only background noise); and unknown commands (e.g. "happy", "Sheila", "cat") for the remaining 20 keyword classes. There are a total of 12 classes: known expression (10 labels), unknown and, silence. An extended version of this dataset [359] was released after publishing the work here described.

As for the architecture, that we name `BinaryCmd` (read as "Binary Command"), we use a stack of three on-the-fly convolutional layers followed by a standard convolution,

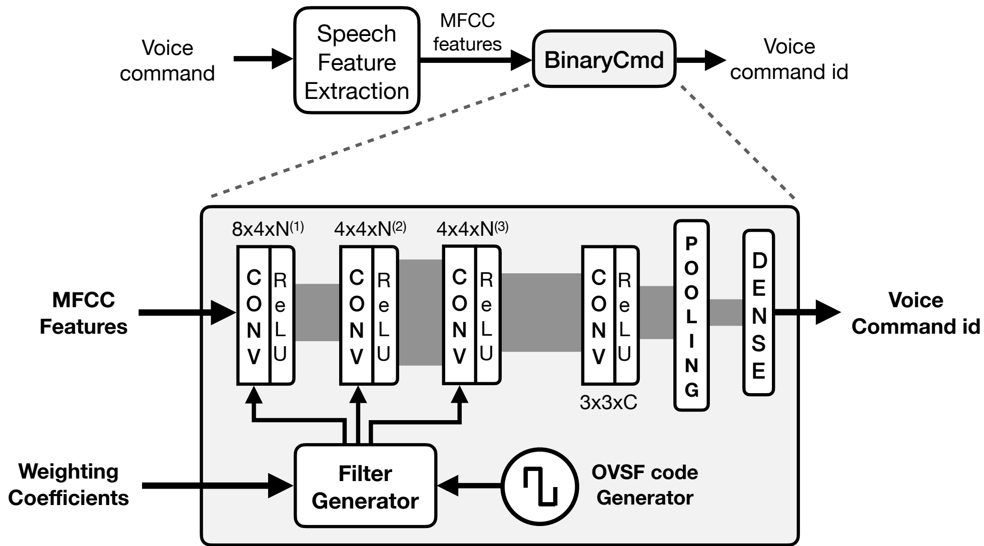


Figure 3.5: (Above) a typical Keyword Spotting system where MFCC features are first extracted from a stream of audio and are then passed to a neural network based classifier. (Below) a diagram showing the main components in `BinaryCmd`. The input convolutinal layer uses 8×4 filters while the next two on-the-fly layers use 4×4 filters. These can be perfectly constructed with OVSF codes since they are a power-of-two. The number of output channels (N) is shown in Table 3.1.

Config	#Filters (N_{out})	Strides $[x, y]$	OVSF Ratios (ρ)	Model Size	OPs
A	$[64, 8, 32]$	$[2, 2]$, $[2, 2]$, $[1, 1]$	$[1.0, 1.0, 1.0]$	24.5 KB	1.8M
B	$[64, 16, 16]$	$[2, 2]$, $[2, 2]$, $[1, 1]$	$[1.0, 0.5, 1.0]$	22.8 KB	2.3M
C	$[16, 16, 16]$	$[2, 2]$, $[1, 1]$, $[1, 1]$	$[1.0, 1.0, 1.0]$	15.8 KB	2.6M

Table 3.1: Parameters for each configuration evaluated for `BinaryCmd` and associated memory and compute footprints. Parameters are given in triplets since there are three on-the-fly convolutional layers (see Figure 3.5). For KWS evaluation, we focus on the accuracy retention when training extremely small models and exclude the compute costs of generating the weights. The number of operations (OPs) for inference do not account for those costs.

pooling and fully connected layers. The input to the KWS system are 49×10 MFFC features extracted from 1 second long audio clips representing each dataset sample. The model outputs a single scalar representing the guessed command id (i.e. one of the 10 words of interest, a silence or, an unknown command). Figure 3.5 shows the main components of the implemented KWS pipeline. We evaluated three variations of such architecture, varying the number of filters, striding and, the OVSF ratio ρ. The parameterization for each model configuration is shown in Table 3.1.

This part of the experimental evaluation was implemented in TensorFlow [2] adding to the source code provided in Zhang et al. [414]. We therefore maintained the use of Adam

Platform	Processor	LUTs	Flip-Flops	DSPs	BRAM
Zynq 7045	Dual-Core Arm Cortex-A9	218,600	437,200	900	2.40 MB
UltraScale+ ZU7EV	Quad-Core Arm Cortex-A53	230,000	461,000	1,728	4.75 MB

Table 3.2: FPGA platforms used for evaluation. The Zynq 7045 is a mid-tier FPGA while ZU7EV can be considered to be a tier higher, both are from Xilinx. On-chip memory are commonly arranged in small blocks (e.g. 36Kb in the case of the Z7045), which can be interconnected or stacked in various ways, hence the term BRAM, which stands for "Block RAM".

optimizer, 30k learning iterations and initial learning rate of 5×10^{-4} and decreased by factors 0.2 and 0.5, after 10k iteration and 20k iterations respectively. 8-bit quantization aware training was used and implemented with Tensorflow's *fake-quantization* layers. We maintained the same dataset splitting ratios with 80% of the commands are used for training, 10% for validation and, the remaining for testing.

3.4.2 Target FPGA Platforms and Baselines

We deploy the larger ResNet-{18,34,50} and SqueezeNet models for ImageNet on two FPGAs platforms in order to asses the speedups obtained from the reduction of off-chip memory accesses when using on-the-fly OVSF-based convolutional layers.

We target two FPGA platforms with different resource characteristics: ZC706 mounting the mid-tier Z7045 and, a ZCU104 with the larger ZU7EV, with a clock rate of 150 MHz and 200 MHz respectively. More details about each platform are shown in Table 3.2. Our hardware designs were synthesized and placed-and-routed with Xilinx Vivado HLS and Vivado Design Suite (v2019.2) and run on both boards. The corresponding Arm CPU was used to set up the off-chip memory transactions, launch the hardware execution and measure the end-to-end performance of each design. In the evaluation, 16-bit fixed-point precision was used. If the instantiated Alpha Buffer cannot store all the $\{\alpha\}_{k=1}^{L}$ describing the model, these would be retrieved from off-chip memory. The available off-chip memory bandwidth was controlled by using a different number of memory ports and amount of word packing, spanning from 1.1 GB/s ($1\times$) to 13.4 GB/s ($12\times$). This allowed us evaluate the idoneity or impact of on-the-fly models at different memory bandwidths.

As for baselines, we introduce two highly optimised single computation engines executing: the vanilla CNNs for ResNet-{18,34,50} and SqueezeNet for ImageNet; and, their pruned variants. For the latter, we use a state-of-the-art method [257] which applies channel pruning based on the first-order Taylor approximation contribution of each filter to the model's loss (i.e. importance-based pruning). This process is carried out iteratively until a target compression ratio is reached. We refer to a pruned model that keeps 82% of the

Model	Accuracy	Memory	OPs	A2S/A2OPs
DS-CNN [414]	**94.4**%	38.6kB	5.4M	2.45/17.48
CRNN [414]	94.0%	79.7kB	3.0M	1.18/31.33
GRU [414]	93.5%	78.8kB	3.8M	1.19/24.6
LSTM [414]	92.9%	79.5kB	3.9M	1.17/23.82
Basic LSTM [414]	92.0%	63.3kB	5.9M	1.45/15.59
CNN [414]	91.6%	79.0kB	5.0M	1.16/18.32
`BinaryCmd`-A	91.4%	24.5kB	**1.8M**	3.73/**50.78**
`BinaryCmd`-B	91.2%	22.8kB	2.3M	4.00/39.57
`BinaryCmd`-C	91.0%	**15.8kB**	2.6M	**5.76**/34.96
DNN [414]	84.6%	80.0kB	**0.16M**	1.05/**528.75**

Table 3.3: Comparison of three `BinaryCmd` configurations against DS-CNN, the current state of the art for KWS applications, and other baselines presented in [414] for MCUs limited to a maximum of 80kB of memory and 6M OPs. Values shown are for networks with 8-bit quantization.

filters as Tay82 and follow the same naming scheme for other ratios. The baseline architecture comprises the same design as the on-the-fly models but with the weights transferred from the off-chip memory into an additional $T_P \times T_C$ buffer, if they do not fit on-chip as in Figure 3.1. Both the vanilla and the pruned baseline models are parameterized with tile sizes $\langle T_R, T_P, T_C \rangle$ and a roofline modeling [407] is used to obtain the highest throughput configuration for the target CNN-FPGA pair during design search exploration.

3.5 Experimental Results

We empirically evaluate models using on-the-fly OVSF-based weights from two perspectives: compression potential and acceleration potential. First, in Section 3.5.1 we demonstrate that OVSF codes can successfully construct competitive models for KWS with extremely low compute and memory footprints. In Section 3.5.2 we evaluate the performance of OVSF models at varying compression ratios, different strategies to construct 3×3 filters and, to select OVSF codes. Then, in Section 3.5.3 we compare the speedups obtained by OVSF models compared to a state-of-the-art structural pruning techniques.

3.5.1 Keyword Spotting with OVSF Models

In Table 3.3 we compare `BinaryCmd` against DS-CNN, a CNN with depth-wise convolutional layers, and all the baselines analyzed in [414]. Our configurations explore the void space of 1M-3M OPs and 10kB-25kB. The results in Figure 3.6, show the potential

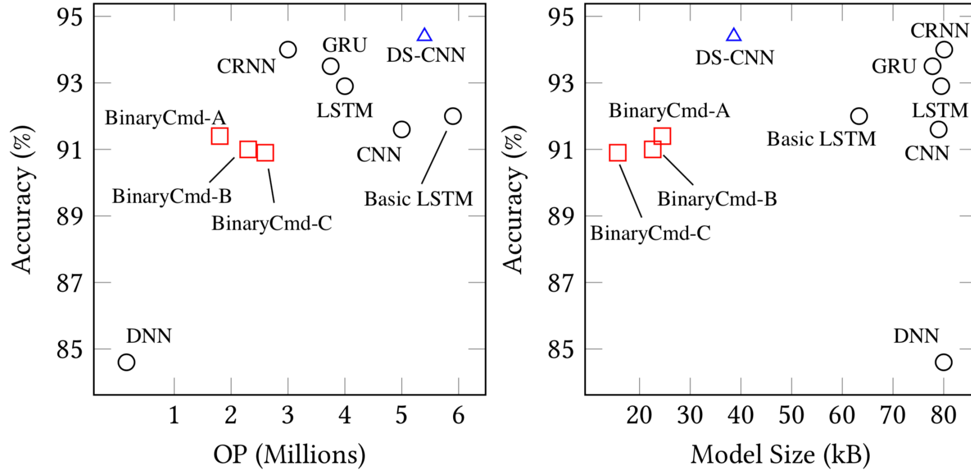


Figure 3.6: Results comparison against architectures in [414] for the category of *small* microcontrollers. DS-CNN has never been tuned below 38.6kB and 5.4M OPs. All other configurations result in larger and computationally more expensive models.

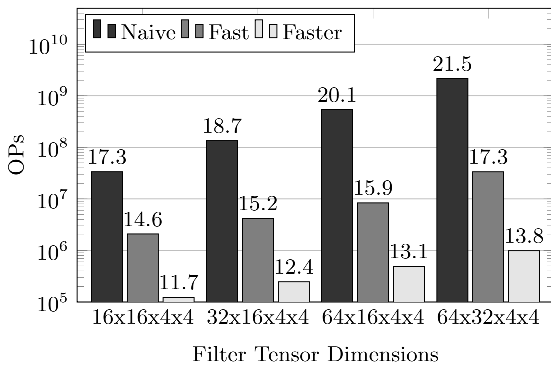


Figure 3.7: Number of OPs required for each filter generation method for different filter dimensions.

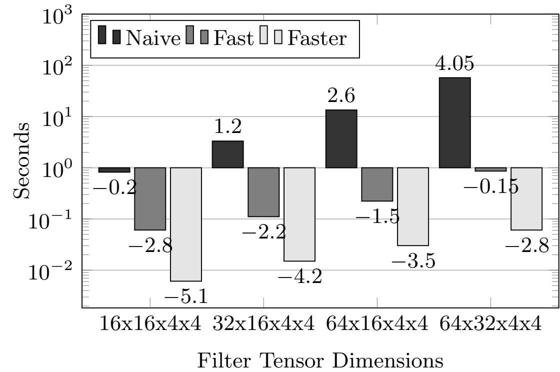


Figure 3.8: Execution time (seconds) required for each filter generation method for different filter dimensions on Arm Cortex-M7.

of our architecture and training formulation: up to 59% model size and 67% number of OPs reduction at the expense of little more than 3% accuracy loss when compared to DS-CNN. All three of our configurations simultaneously achieve top *accuracy-to-size* (A2S) and *accuracy-to-OPs* (A2OPs) ratios meaning that `BinaryCmd` is a good first step towards the design of architecture capable of providing over 90% accuracy levels with minimal memory footprint and low computational costs.

When comparing against the other baselines in terms of the compute budget required to run inference, we have excluded the costs of generated the OVSF-based filters on-the-fly. As it was first discussed in Section 3.3.3, our on-the-fly formulation can be efficiently implement with custom hardware, `CNN-WGen`, allowing for full control on memory hi-

erarchies, data paths and, logic. We argue that CPUs and MCUs can still benefit from the on-the-fly formulation with OVSF codes if a co-processor implementing `CNN-WGen`'s functionality is available. Nonetheless, we measured the compute in terms of number of operations (OPs) and latency costs of generating filters of different shapes including some of the ones found in the three variants of `BinaryCmd`. These are shown in Figure 3.7 and Figure 3.8 respectively. The platform used is a SRM32F767ZI Arm Cortex-M7 with 512kB of SRAM and 2MB of flash. These measurements use a lightly optimised OVSF code generator written in C and using 32-bit weights and weighting coefficients. The *naive* approach performs a truncated combination of $\hat{L} = \lfloor \rho \cdot L \rceil$ tensors with shape $N_{out} \times N_{in} \times K \times K$, which is very computationally expensive. Both *fast* and *faster* were described in 3.3.2: the former performs a concatenation of N_{out} tensors of shape $N_{in} \times K \times K$, resulting in $\mathcal{O}(N_{out}(N_{in}KK)^{\hat{L}})$ and 860ms to generate the largest filter considered; the latter concatenates $N_{out}N_{in}$ matrices of shape $K \times K$, of complexity $\mathcal{O}(N_{out}N_{in}(KK)^{\hat{L}})$. For this approach, generating a $64{\times}32{\times}4{\times}4$ weights tensor took 60ms. To put these costs in context with the baseline methods shown in Table 3.3, the total costs of generating the three OVSF-based layers in `BinaryCmd`-{A,B,C} ranged between 10ms and 14ms. This represents around 10% of the total latency of the state-of-the-art DS-CNN model which required 131ms on the same Cortex-M7 MCU [27].

The main takeaways from this first evaluation of on-the-fly models with OVSF codes on the lightweight but challenging task of KWS are two fold: OVSF-based models can compete with other type of layers while using fewer parameters and operations during inference despite requiring a more complex training procedure; second, the costs associated to the construction of filters, even for models with small memory footprint, is not negligible. This last point motivated the exploration of ways to accelerate this workload, resulting in the weights generator, `CNN-WGen` and, the rest of the `unzipFPGA` infrastructure.

3.5.2 Image Classification with OVSF Models

In Table 3.4 we compare the performance of on-the-fly OVSF models for image classification on CIFAR-10 at different compression ratios, methods for extracting $3{\times}3$ filters and, strategies for selecting OVSF codes when $\rho{<}1$. First, we evaluate on ResNet-18 and ResNet-34 following the same architecture that will later be used for ImangeNet. The only difference is the replacement of the final fully connected layer to match the number of classes, as well as the input convolutional layer to accommodate for the smaller input size of $3{\times}32{\times}32$ instead of $3{\times}224{\times}224$. For OVSF100, OVSF75 and OVSF50, both architectures maintain their respective accuracy levels regardless of the basis selection strategy or method to extract a $3{\times}3$ filter. Only when seeking higher compression ratios with OVSF25

Model Arch. (baseline)	Basis Strategy	Filters to $3{\times}3$	OVSF100 Param.	OVSF100 Acc.	OVSF75 Param.	OVSF75 Acc.	OVSF50 Param.	OVSF50 Acc.	OVSF25 Param.	OVSF25 Acc.	OVSF12.5 Param.	OVSF12.5 Acc.
ResNet18 (93.2% 11.2M)	Sequential	Crop	19.7	93.9	10.6	93.9	9.1	93.7	3.6	92.9	2.9	**91.8**
		Adaptive		93.7		93.9		93.8		93.0		91.5
	Iterative	Crop		94.1		93.6		93.6		93.6		**93.6**
		Adaptive		94.0		93.7		93.8		92.3		90.4
ResNet18† (91.3% 0.27M)	Sequential	Crop	0.48	90.8	0.27	90.9	0.25	90.8	0.15	88.3	0.1	**83.0**
		Adaptive		91.1		91.2		91.2		88.5		81.8
	Iterative	Crop		91.1		91.3		91.3		91.4		90.4
		Adaptive		91.2		91.4		91.4		91.0		90.4
ResNet34 (93.9% 21.3M)	Sequential	Crop	37.7	94.1	20.2	93.7	17.6	93.9	7.2	93.4	5.2	**92.0**
		Adaptive		94.3		93.9		94.0		93.4		91.8
	Iterative	Crop		94.1		93.6		93.8		94.3		**93.9**
		Adaptive		93.8		93.8		93.7		93.2		91.4
ResNet34† (92.1% 0.46M)	Sequential	Crop	0.82	92.3	0.54	91.7	0.43	91.4	0.26	89.3	0.16	**84.1**
		Adaptive		92.2		91.5		91.5		89.2		82.4
	Iterative	Crop		92.3		92.2		91.8		92.2		**91.2**
		Adaptive		92.4		92.1		91.7		91.7		91.1

Table 3.4: Analyzing the impact on accuracy for each basis selection strategy (relevant when $\rho{<}1$) and method to obtain $3{\times}3$ filters from $4{\times}4$ OVSF filters. Models trained on CIFAR-10 on the commonly used ResNet-18/34 implementations for this dataset. We also test on the much smaller variation of such architectures (†) proposed by He et al. [146]. Performing an iterative drop of basis, as opposed as taking the first $\lfloor \rho \cdot L \rceil$, consistently results in better models. As model size is reduced, taking a $3{\times}3$ crop from a $4{\times}4$ filter performed better than using an average pooling stage.

and OVSF12.5, we observe that performing a greedy selection of OVSF codes as opposed of taking the first $\lfloor \rho \cdot L \rceil$ works better. In addition, taking a $3{\times}3$ crop out of a $3{\times}3$ filter shows up to 3 percentage points (pp) improvement compared to using an adaptive average pooling for this purpose. This becomes evident in the OVSF12.5 settings, where we aggressively compress the last layers of the networks by discarding 87.5% of the basis.

For the much smaller ResNet-18† and ResNet-34†, that are around $37\times$ smaller than the commonly used version of these architectures, the impact of the approach followed to choose which OVSF basis to utilize is more dramatic. Using an iterative approach results in 8.6 pp and 8.8 pp higher accuracy for ResNet-18† and ResNet-34† respectively. Regarding the extraction of $3{\times}3$ filters, taking a crop also resulted in the best option in general.

Informed by the analysis on the CIFAR-10 dataset and results in Table 3.4, the experiments on ImageNet were configured to take a $3{\times}3$ crop and follow an iterative greedy stage for selecting the OVSF basis. Results are shown in Table 3.5 for ResNet-18, in Table 3.6 for ResNet-34 and, in Table 3.7 for SqueezeNet. For ResNets, OVSF50 shows better performance than Tay82 and ResNet18-OVF25 is as accurate as ResNet18-Tay82 while requiring about half the number of model parameters. ResNet34-OVSF25 yields 3.7 pp higher accuracy than ResNet34-Tay56, despite using 25% fewer parameters. The benefits in terms of throughput of using OVSF on-the-fly models are discussed in the next subsection.

Model Arch.	Compression Method	Params (millions)	Accuracy (%)	Performance (inf/sec) ($1\times$, $2\times$, $4\times$)
ResNet-18	-	11.7	69.8	(12.0, 23.5, 40.1)
ResNet-18	Tay88	9.1	68.8	(14.3, 28.0, 46.4)
ResNet-18	Tay82	7.9	67.3	(14.3, 27.8, 45.4)
ResNet-18	Tay72	6.0	64.8	(18.2, 35.3, 57.6)
ResNet-18	Tay56	3.7	58.3	(23.8, 47.3, 82.2)
ResNet-18	OVSF50	9.1	69.2	(19.4, 33.8, 49.9)
ResNet-18	OVSF25	4.1	67.3	(19.4, 34.8, 51.0)
ResNet-18	Tay82 + OVSF50	6.3	66.2	(24.5, 43.2, 57.9)
ResNet-18	Tay82 + OVSF25	2.8	64.4	(24.5, 43.6, 59.7)

Table 3.5: Accuracy and number of parameters for ResNet-18 models on ImageNet following different compression schemes. Performance measured on the Zynq 7045 platform at different memory bandwidths.

3.5.3 Reducing Data Movement with OVSF Models

We now assess the actual performance gains of `unzipFPGA` compared to optimized ResNet-18, ResNet-34 and SqueezeNet baselines for deployment on the Zynq 7045 and Ultra-Scale+ ZU7EV platforms. Tables 3.6 and 3.5 show the achieved validation set accuracy and actual performance of each design as measured on ZC706 under varying bandwidth budget. Across bandwidths ($1\times/2\times/4\times$ where $4\times$ is the 4.5 GB/s peak measured bandwidth on ZC706), `unzipFPGA`'s OVSF50 and OVSF25 designs outperform the faithful baseline. As bandwidth availability increases, the baseline becomes less memory-bound and the performance gap w.r.t `unzipFPGA` models closes. Table 3.7 shows the comparison of `unzipFPGA` with the faithful baseline for SqueezeNet on ZU7EV with peak measured bandwidth of 13.4 GB/s ($12\times$). Both OVSF50 and OVSF25 designs yield increasing throughput gains as the bandwidth becomes more restricted, with OVSF25 sustaining over 57% speedup for up to $4\times$ bandwidth.

Across architectures, at $1\times$ bandwidth, OVSF25 offers minimal additional gains despite resulting in smaller memory footprints. This is because, below a compression ratio, even though the memory accesses and utilization for model parameters are further reduced, the activations begin to dominate I/O, and hence further weights reduction does not provide significant benefits.

Compared to the pruned baselines, `unzipFPGA`'s OVSF models are more resilient at high compression ratios while resulting in similar accuracy at lower compression ratios. In terms of throughput, `unzipFPGA` delivers faster processing at more constrained bandwidths. Concretely, ResNet-34-OVSF50 is 80% faster than Tay82 at $1\times$ bandwidth, with

Model Arch.	Compression Method	Params (millions)	Accuracy (%)	Performance (inf/sec) (1×, 2×, 4×)
ResNet-34	-	21.8	73.3	(8.6, 16.8, 28.7)
ResNet-34	Tay82	17.4	72.7	(10.7, 21.0, 35.6)
ResNet-34	Tay72	15.1	71.9	(13.3, 25.8, 44.0)
ResNet-34	Tay56	9.4	67.8	(18.3, 36.3, 63.8)
ResNet-34	Tay45	6.3	63.1	(21.8, 43.4, 79.8)
ResNet-34	OVSF50	17.2	72.8	(18.1, 21.8, 31.1)
ResNet-34	OVSF25	7.2	71.5	(18.4, 27.3, 33.5)
ResNet-34	Tay82 + OVSF50	13.2	71.1	(18.6, 30.0, 37.3)
ResNet-34	Tay82 + OVSF25	6.7	70.6	(18.8, 31.0, 38.9)
ResNet-34	Tay72 + OVSF50	11.9	70.3	(18.8, 32.0, 40.2)
ResNet-34	Tay72 + OVSF25	4.9	68.9	(18.9, 33.3, 42.0)

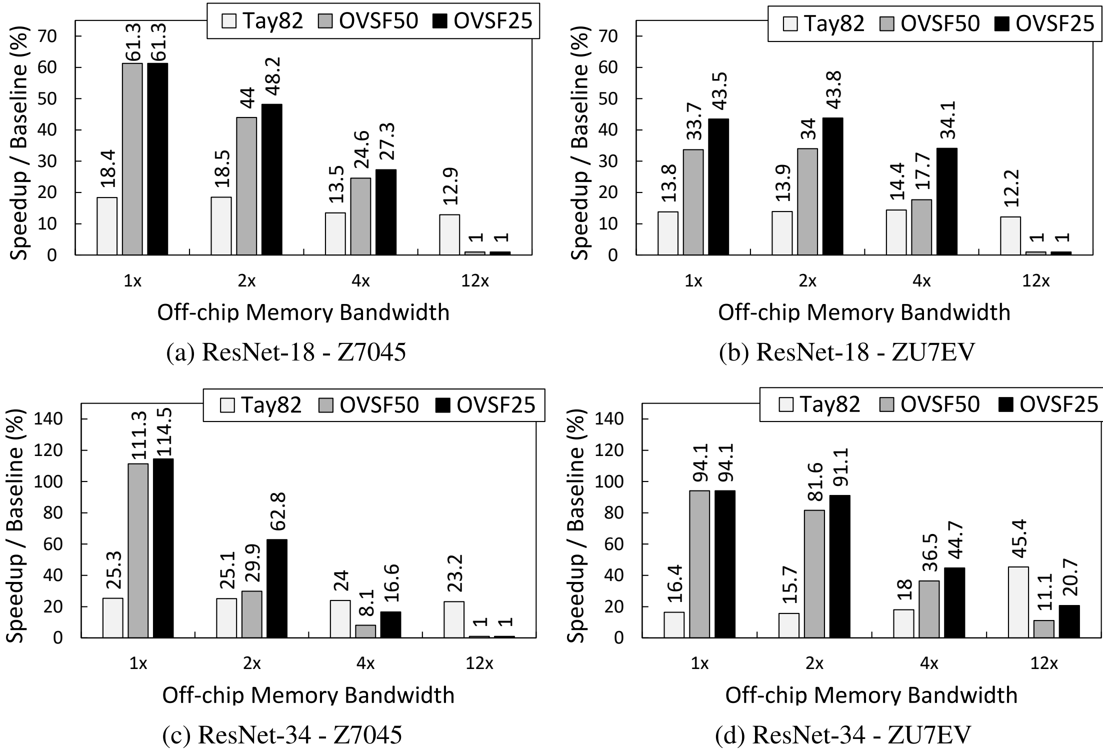
Table 3.6: Accuracy and number of parameters for ResNet-34 models on ImageNet following different compression schemes. Performance measured on the Zynq 7045 platform at different memory bandwidths.

Model Arch.	Compression Method	Params (millions)	Accuracy (%)	Performance (inf/sec) (1×, 2×, 4×, 12×)
SqueezeNet	-	1.24	58.2	(54.7, 108.9, 217.8, 515.6)
SqueezeNet	OVSF50	1.07	57.6	(97.4, 189.7, 339.1, 594.1)
SqueezeNet	OVSF25	0.86	57.1	(97.4, 189.7, 342.6, 600.5)

Table 3.7: Comparing with faithful baseline on SqueezeNet on ImageNet. Performance measured on the UltraScale+ ZU7EV platform at different memory bandwidths.

less than 1pp accuracy drop. Despite being almost identical in terms of model size and accuracy, Tay82's approach, which prioritizes the pruning of layers with the least accuracy impact, leads to the pruning of mostly compute-bound layers when targeting ResNet-34. On the other hand, ResNet34-OVSF50 compresses more effectively memory-bound layers, leading to significantly higher throughput at low bandwidths. A similar pattern is observed for ResNet-18.

To explore the benefits of combining `unzipFPGA`'s OVSF execution scheme with channel pruning, we derive, train and map Tay-OVSF models. This approach yields competitive lightweight models that are not attainable through structural pruning alone. For instance, ResNet-18 with Tay82+OVSF25 is 25% smaller than ResNet-18-Tay56 and achieves 6.1 pp higher accuracy, while achieving 34.6% and 23.5% higher throughput over ResNet-18-Tay72 with less than 0.5 pp accuracy drop. Other activation compression techniques [60, 270] can be orthogonally combined to obtain further gains. In Appendix A.3 we compare all the architectures considered in this section with existing FPGA designs.



(a) ResNet-18 - Z7045 (b) ResNet-18 - ZU7EV

(c) ResNet-34 - Z7045 (d) ResNet-34 - ZU7EV

Figure 3.9: Speedup over optimized ResNet-18 (a-b) and ResNet-34 (c-d) baselines when varying the available off-chip memory bandwidth. On-the-fly models offer much faster inference stages in the context of restricted bandwidth. This gap compared to Tay82, which does structured pruning, becomes even larger on the Z7045, a lower-end FPGA platform. In light of these results, the on-the-fly formulation presented in this chapter becomes a good candidate for supporting multi-tenant FPGA platforms.

3.6 Discussion

This chapter has presented the first formulation of on-the-fly models, enabling model acceleration by reducing data movement. In this section we highlight some of the limitations of the on-the-fly models presented in this work as well as some future directions for this new compression paradigm.

Assigning ρ based on layer boundness. Across all works that collectively form this chapter [324, 102, 333], we maintained the formulation of each of the different OVSF ratios, specifically OVSF-{100,75,50,25,12.5}, which were originally defined in [324]. As presented in detail in Section 3.4, deeper layers in the networks are compressed more aggressively. This compression ratio to layer mapping often translated into memory bound layers being assigned higher ratios. However, as newer architectural designs become more complex, it is no longer straightforward to determine which layers would be memory bound. Can we inform the assignment of each layer's ρ parameter from a pre-training stage in which we measure each layer's level and type of boundness? Can we automate

the discovery of each layer's optimal ρ at training time to better capture the impact on performance (e.g. accuracy) for compressing each layer?

Using better codes. In this first on-the-fly formulation we relied on OVSF codes due to their properties of being easy to generate (even for very large dimensions) in a deterministic manner. However, their simplicity also introduced limitations. Having to treat these codes as a basis (in the Algebra sense) of $\mathbb{R}^L$ in conjunction of the method followed to perform a linear combination, unavoidably introduced lower-bound ρ setting a limit on how much OVSF-based models can realistically be compressed. Can we take advantage of the recent advancements in NAS to automate the discovery of codes that are (1) easy to generate on-the-fly, (2) deterministic and, (3) expressive enough so only a few of them are needed to represent large portions of the weights tensor?

Multi-tenant applications. The on-the-fly formulation for image classification models and the approach to generating them using custom hardware with `CNN-WGen`, demonstrated superior performance to structured pruning in settings with restricted memory bandwidth. This can better be observed in Figure 3.9. This is precisely the scenario where on-the-fly models have the higher impact potential since their core aim is to accelerate inference by reducing data movement from off-chip memory and not, at least not necessarily, by doing fewer operations, as pruning seeks to achieve. This characteristic of on-the-fly models would support the deployment of multiple applications running concurrently on the same FPGA while each sharing off-chip memory resources, accessed at either $1\times$ or $2\times$. For example, performing audio-visual scene understanding by means of two models running concurrently: an object segmentation network and an audio system performing source separation and spatial localization. This would be harder to achieve with alternative approaches to reduce model size while retaining high accuracy, such as structural pruning.

3.7 Summary

This chapter has presented a first formulation for on-the-fly models aimed at accelerating inference by reducing the latency costs associated to data movement. This has been achieved by learning a compressed representation of the model parameters leveraging OVSF codes and, implementing a lightweight decompression stage by which the model parameters are materialized ahead of computation only. Such representation is compact, keeping the cost of instantiating the model parameters low and without outweighing the latency savings of performing fewer accesses to off-chip memory or DRAM. In this chapter

we also derived and implement `CNN-WGen`, a hardware weights generator for OVSF-based models capable of sustaining high processing rates without slowing down layer inference. The experimental evaluation in this chapter showed that: OVSF codes can be used to compress small and large models with little degradation for both image classification and keyword spotting applications; we considered different ways to selecting OVSF codes for $\rho<1$ as well as two strategies to extract 3×3 filters; finally, we showed how models derived and mapped using `unzipFPGA` outperform both status-quo and pruned CNN engines for the same bandwidth. A ResNet-34 trained with OVSF50 is up to $1.7\times$ faster than its structurally pruned counterpart while both achieving the same accuracy on ImageNet. The benefits of the formulation for on-the-fly models presented in this chapter manifest more clearly at lower memory bandwidths, suggesting that multiple on-the-fly models could co-exist in a given FPGA without slowing down each other when accessing off-chip memory.

Chapter 4

Algorithms and Architecture Search for Lightweight Inference

Inference efficiency is crucial for deployments on mobile and IoT devices that are often constrained in terms of memory and compute and, are battery powered. Over the years, multiple approaches have been proposed to alleviate the compute-bound nature of convolutions. Among the various techniques earlier presented in Section 2.1, the use of lightweight architectural designs and the use of quantization stand out as the *de facto* choices for CNNs. In this way, the design of lightweight CNNs is often reduced to the manual or automated discovery of (1) the minimal compute graph and, (2) the lowest bitwidth for each node's operands that meet the application's performance criteria. These two optimization dimensions have enabled the deployment of sophisticated CNN models on constrained devices.

A third dimension, orthogonal to lightweight architectural designs and quantization, considers the choice between different convolution algorithms for each layer resulting in either faster inference, lower resource utilization, or both. Each algorithm comes with its own trade-offs [14], but in this chapter we focus on the Winograd algorithm since it is the fastest known algorithm for convolutions of the dimensions often found in CNNs. However, despite this being widely known by the research community, Winograd convolutions are never found in deployments of models that make use of quantization and instead, other slower algorithms such as `im2row` or `im2col` are utilized. The reason for this is the inherent numerical error of the Winograd algorithm which is exacerbated as bitwidth is reduced, among other factors, collectively rendering models relying on this algorithm unusable due to a very severe accuracy degradation.

Research Question: *Can we capture such errors during training to learn models that are aware of their inherent limitation but remain accurate? Can we combine integer arithmetic with the fast Winograd algorithm to obtain even faster convolutional layers?*

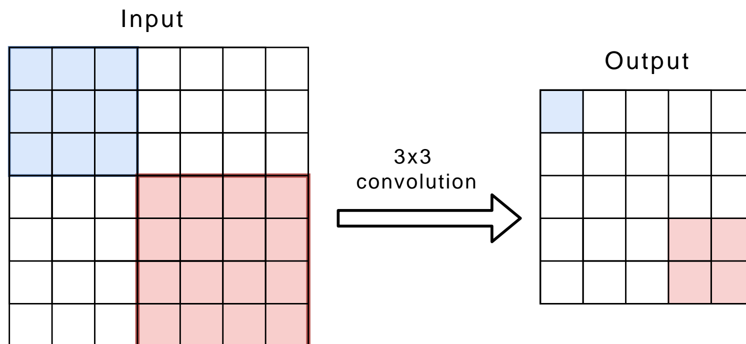


Figure 4.1: Standard convolutions operate on a tile (blue) defined by the filter size, generating a single output. Winograd convolutions operate on larger tiles without modifying the filter dimensions. A 3×3 Winograd convolution operating on a 4×4 tile (red) generates a 2×2 output. This is often expressed as $F(2\times2, 3\times3)$.

4.1 Problem Setting and Contributions

The Winograd convolution performs the bulk of the computation as a Hadamard product between weights and input in the Winograd space requiring $\mathcal{O}(n)$ operations. Unlike normal convolutions, that generate a single output per convolution, a Winograd convolution computes several outputs simultaneously. This property makes Winograd convolutions minimal in the number of *general multiplications*[1][368]. While normal convolutions operate on tile sizes matching the width and height of the filter, Winograd convolutions can operate on larger tiles. This is illustrated in Figure 4.1. In this way, while normal convolutions would require 8.1K multiplications to densely convolve a 32×32 input with a 3×3 filter, Winograd convolutions operating on a 4×4 tile require only 3.6K. The speedups Winograd convolutions offer increase with tile size.

However, exploiting the speedups that Winograd offers exposes a problem inherent in current Winograd convolution implementations: numerical error. This error, which grows at least exponentially [268] with tile size, is the primary reason why Winograd is generally only deployed with 32-bit floating point and for comparatively small tile sizes, rarely larger than 6×6. In practice, an architecture with standard convolutional layers would first be trained before replacing standard convolution with Winograd convolution for deployment.

In this chapter, we focus on alleviating the problem of numerical error that arises when using Winograd convolutions in quantized neural networks. Achieving this ultimately enables us to combine the speedups of Winograd with those that reduced precision arithmetic is known to offer, among other benefits in terms of energy and area. To this end, in Section 4.4 we present an end-to-end training pipeline that exposes the numerical inaccuracies

[1]*General multiplications* is a term commonly used in Winograd jargon referring to element-wise or Hadamard product stage.

Figure 4.2: The `wiNAS` framework brings all the contributions presented in this chapter together. It leverages Winograd-aware layers that capture the numerical errors of Winograd convolutions during training. It uses a real latency model to guide the neural architecture search (NAS) stage which, does not change the model architecture, but jointly optimizes the choice of convolution algorithm for each 3×3 layer balancing inference speedup and numerical degradation. `wiNAS` automates this process based on the target platform and task at hand. On the right, the resulting ResNet-18 macro-architecture is shown. These diagrams are better shown in Figure A.2 in the Appendix, with `wiNAS` $_Q$ and WA_{F4} being described in Section 4.5.1.

introduced by Winograd to the learning of the model parameters. We also address the source of the numerical error and propose a relaxation on the form of the transformation matrices used in the Winograd convolution algorithm. We achieve this by adding them to the set of *learnable* parameters in a layer, after initializing them via Cook-Toom [201].

Finally, we introduce `wiNAS`, a Winograd-aware Neural Architecture Search framework which leverages Winograd-aware layers and latency measurements on Arm Cortex-A73 and A53 cores, to jointly optimize for high accuracy and low latency. This framework transforms a given macro-architecture by replacing each 3×3 convolution with either `im2row` or Winograd convolutions of different tile sizes. Figure 4.2 illustrates these contributions in a compact manner. Before proceeding with the core contributions of this chapter, Section 4.2 provides an overview of other works making use of Winograd convolutions, the source of their numerical degradation and, existing hardware-aware NAS frameworks.

4.2 Background and Related Work

Convolutions are the *de facto* spatial feature extractor in neural networks. As a result, a number of approaches have emerged to reduce the computational costs of using this operator. These include: compact implementations of convolutional layers, reduced precision arithmetic, new data formats or, the use of alternative formulations of the convolution operation. The background relevant to the former set of techniques and others was introduced in Section 2.1. In this section, we delve into how the Winograd algorithm for convolutions has made its way into modern CNNs, the sources of numerical degradation that have prevented Winograd convolutions from being used in deployments using integer-only arithmetic and, how hardware models have been integrated into NAS frameworks to guide the discovery of hardware-aware architectures. This will be relevant when introducing `wiNAS`.

4.2.1 Winograd Convolutions in modern CNNs

The Winograd algorithm for convolutions was first applied to CNNs by Lavin & Gray [207], showing $2.2\times$ speedup compared to cuDNN [62] on a VGG [305] network, with no loss in accuracy on small 4×4 tiles and batch one. However, exploiting Winograd convolutions on larger input tiles is challenging due to numerical instability. This point is addressed in Section 4.2.2. Prior to the work presented in this chapter, several works studying the suitability of Winograd convolutions in memory and compute constrained setups were proposed. These include: the use of integer arithmetic for complex Winograd convolutions [249]; a general formulation for the Winograd algorithm [31] that shows promising results in FP16 and BFLOAT16 when using higher degree polynomials, as opposed to linear (degree=1) polynomials; an efficient region-wise multi-channel implementation of Winograd convolutions using GEMMs [241] that achieves $4\times$ speedups on Arm Cortex-A CPUs; and, a technique [232] that enables up to 90% sparsity in the Hadamard product stage of the Winograd algorithm, effectively reducing by $10\times$ the number of multiplications with no accuracy loss in FP32 models. This was achieved by applying ReLU and pruning in the Winograd domain to the input and weights tensors respectively.

More recently, several works have presented solutions to some of the limitations of the Winograd algorithm for convolutions. These include: a method [169] that enables the use of strides larger than one when using larger filters by decomposing the convolution into smaller 3×3 Winograds of stride one, which in turn also translates into smaller numerical degradation; an alternative solution to the same problem was presented by Jingbo

et al. [182], which takes advantage of data dependencies in contiguous tiles when reformulating the Winograd convolutions in a nesting decomposition[2] form; or a framework that proposes the acceleration of Winograd convolutions by performing the Hadamard product stage using integer arithmetic [216], while the model still uses FP32 weights and activations. More recently, Li et al. [222] made use of a Winograd-like implementation which replaces the element-wise Hadamard stage with a set of matrix additions, offering further speedups and energy savings. A follow-up work to the one presented in this chapter made use of Legendre polynomials to construct the Winograd transformation matrices [29] and showed better performance when using these to initialize Winograd-aware layers.

4.2.2 Numerical Degradation in Winograd Convolutions

Unlike alternative implementations of the convolution algorithm (e.g. `im2row`, and others presented in Section 2.3) that operate directly on the input and weights tensors or replicated versions of these, the Winograd algorithm for convolutions operates on a polynomial transformation domain: the *Winograd domain*. Operating on this domain, requires the linear transformation of inputs and weights (*pre-processing* stage), perform a Hadamard product (*convolution* stage) and, transform the result back to the original domain (*post-processing* stage). This three-stage process is mathematically represented in Equation (4.1), which will be presented in greater detail later. Each of the three matrix-matrix multiplications and the element-wise multiplication, are susceptible to rounding error, and its accumulation after each stage [154, 121]. The error bound for Equation (4.1) is proportional to three terms: the sum of rounding error introduced when performing the linear transformations; the product of norms, $\|\cdot\|_1$, of of input and weight tensors; and, the product of norms of transformation matrices G, $B^\top$ and, $A^\top$ [30]. This last term, which dominates the numerical degradation, grows at least exponentially with the size of the convolution [268]. The source of this is the ill-conditioned nature of the Vandermonde matrices—from which G, $B^\top$ and, $A^\top$ derive—, where rows coefficients follow a geometric progression and are constructed from a set of so-called *polynomial points*. These are implementation specific, meaning that different set of points are needed depending on the filter and input sizes used in the convolution.

The challenge is to select a set of polynomial points that minimizes the difference between: the output of Equation (4.1), which is fast to compute but potentially highly inaccurate; and, that of a direct convolution, which is slower than Winograd convolutions in most cases but accurate as it is only susceptible to standard FP rounding error. On this front, a few works stand out and propose: constructing transformation matrices using trimmed

[2]See Section 11.1 in Blahut [37]

Vandermonde matrices and a triplet of scaling matrices that minimize matrix norms and degradation as a result [335]; a modified Cook-Toom [201] algorithm for constructing the transforms, which reduces the rate at which the numerical degradation grows [30] with tile size; and more recently, the work of Liu & Mattina [235], where the Winograd convolution is performed using the Residue Number System (RNS) [254], allowing for a more numerically stable representation of Vandermonde matrices using integers and therefore facilitating their use in INT16 and INT8 convolutions.

In this chapter we present a data-driven relaxation on the form of transformation matrices G, $B^\top$ and, $A^\top$. This relaxation contributes to the easing of the numerical instability known to affect Vandermonde matrices, which becomes more severe when operating in reduced-precision spaces and when making use of larger tile sizes.

4.2.3 Hardware-aware Neural Architecture Search

Early forms of Neural Architecture Search (NAS) fixated in finding architecture designs that minimize high level metrics such as model size or number of operations (FLOPs) needed to perform inference [42, 318, 32, 230]. However, FLOPs is only a proxy metric of the latency of a model and, as a result, it is often not possible to compare two models with similar number of FLOPs and claim one is faster than the other [203]. Similarly, model size accounts only for a fraction of the memory needed during inference and does not account for other sources of memory consumption such as activations or intermediate tensor due to layer branching (as in residual connections). Furthermore, the same model and implementation (e.g. choice of convolution algorithm, as discussed in Section 2.3) might be optimal when running on some hardware but become severely bottlenecked on another due to, among others, asymmetries in memory hierarchy and bandwidth and, overall compute capabilities of the target device. Nevertheless these early attempts paved the way for more sophisticated frameworks that do introduce various forms of hardware-awareness (e.g. latency, memory peak, energy consumption, etc) into the NAS stage. Among these are: ProxylesNAS [47], which constructs a latency model after measuring latency for all the candidate operators in the search space for a given target hardware. Then, during search, a latency-based term is added to the loss, penalizing the choice of slower layers; BRP-NAS [90] proposed the use of a latency predictor implemented as a Graph Convolutional Network (GCN) that eases the challenge of obtaining accurate and abundant latency metrics on the target platform/s and, more accurately predicts a model's latency than other methods [47] relying on an aggregate of layer-wise metrics; more recently, Abdelfattah et al. [5] propose the use of *zero-cost* proxies which by performing forward-backward propagation on a single batch, an accurate estimate of the model's accuracy can be obtained, enabling

the ranking of thousands of candidate architectures very efficiently; other frameworks such as SpArSe [99] and μNAS [225] target MCUs and consider not only latency but also memory peak and, operation ordering during the search process. Other works have also consider introducing energy consumption into the search space [166, 122], both in terms of peak and average energy consumption.

The work presented in this chapter leverages NAS to find the optimal convolution algorithm (i.e. `im2row` or different Winograd implementations) for each layer in the model while preserving the overall network macro-architecture and model size. This will be introduced later in Section 4.4.2, when presenting `wiNAS`.

4.3 The Winograd Algorithm for Convolutions

This section introduces Winograd convolutions and their trade-offs in terms of compute, memory and accuracy. Then, in Section 4.4 we present the Winograd-aware layers used in our networks, their advantages and, how they get integrated into `wiNAS`.

The Winograd algorithm for convolutions using linear polynomials guarantees to use the minimum number of element-wise multiplications to compute $m \times m$ outputs using an $r \times r$ filter. Lavin & Gray [207] refer to this minimal algorithm as $F(m \times m, r \times r)$ and present its matrix form as:

$$Y = A^\top \Big[[GgG^\top] \odot [B^\top dB] \Big] A \tag{4.1}$$

$$A^\top = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}, \quad B^\top = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad G = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

where G, B and A are transformation matrices applied to the filter g, input and output respectively and $\odot$ is the Hadamard or element-wise multiplication. In a multi-channel convolution, the result of the Hadamard stage would normally be first summed across channels and then transformed with A. The transformation matrices shown above are default for $F(2{\times}2, 3{\times}3)$ Winograd convolutions.

These transformation matrices are commonly constructed[3] as described in the Cook-Toom algorithm which requires choosing a set of so-called *polynomial points* from $\mathbb{R}^2$. This choice is not trivial, but for small Winograd kernels e.g. $F(2{\times}2, 3{\times}3)$ or $F(4{\times}4, 3{\times}3)$,

[3] See Section 5.2 in Blahut [37] for a step-by-step example.

there is a common consensus on which set of points to choose. While a standard convolution using a $r \times r$ filter g would operate on a $r \times r$ input tile, a Winograd convolution expressed as Equation (4.1) expects an input patch d with dimensions $(m{+}r{-}1) \times (m{+}r{-}1)$. The key difference is that while the standard convolution would generate a 1×1 output, the Winograd convolution would compute a $m \times m$ output. In this way, a standard 3×3 convolution requires 9 multiplications per output (*mpo*), $F(2{\times}2, 3{\times}3)$ and $F(4{\times}4, 3{\times}3)$ require 4 *mpo* and 2.25 *mpo* respectively. This was first visualized in Figure 4.1. Theoretically, these savings grow as we increase the tile or filter sizes. For the remaining of this work and, unless stated otherwise, we will be considering only 3×3 filters and therefore refer to $F(2{\times}2, 3{\times}3)$ as $F2$, $F(4{\times}4, 3{\times}3)$ as $F4$, and so on.

Having introduced the Winograd algorithm for convolutions and its notation, we now enumerate the challenges associated with the use of such algorithm in modern CNN-based models. These span three dimensions:

Compute. Winograd convolutions require the transformation of both tile d and filter g to the Winograd domain. The cost of these transformations grows quadratically with m, and can represent a significant portion of the total computation of up to 75% (Section 4.6.2). This suggests that Winograd offers little to no speedup in layers with few filters, as it is the case in the first few layers of most CNNs. The cost of $GgG^\top$ is often ignored as it is amortized across inferences assuming it can be pre-computed in advance.

Memory. In Equation (4.1), $GgG^\top$ transforms the filter g to the Winograd domain, matching the dimensions of the input tile d. This results in an increase of run-time memory associated with the weights: $1.78\times$ and $4\times$ for $F2$ and $F4$ respectively for 3×3 filters. The rate at which memory increases is a function of the filter size as shown in Figure 4.3 This is especially undesirable for memory-constrained devices such as microcontrollers or low-end mobile CPUs.

Numerical Error. Small $F2$ and $F4$ perform well in single and double precision (FP32/64) networks and are available in production-ready libraries such as cuDNN [62] and Arm Compute Library [19]. Because these introduce only marginal numerical error, a network can first be trained using conventional convolutions before replacing them with Winograd for deployment, without impacting accuracy. However, attempting this with larger Winograd tiles, or in combination with quantization, results in significant accuracy loss. In practical terms, the root of the problem is the increasing numerical range in G, B and A as d grows. As a consequence, the multiple matrix multiplications in Equation (4.1) introduce considerable numerical error, ultimately reducing accuracy. This problem is exacerbated in networks using quantized weights and activations, where the range and precision of values is reduced. We show these limitations in Figure 4.4, where we compare the

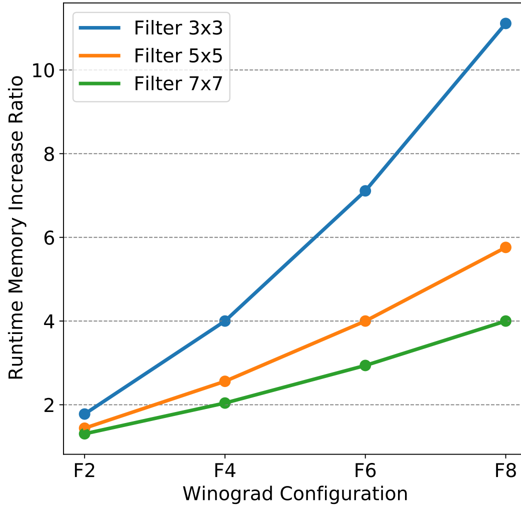


Figure 4.3: When transforming the filters to the Winograd domain, $GgG^{\top}$, these are made to match the input tile size. Larger tiles sizes offer greater parallelization potential at the cost of increased run-time memory usage. Smaller $3{\times}3$ filters, ubiquitous in modern CNNs, disproportionately suffer more from increasing tile sizes than $5{\times}5$ and larger filters.

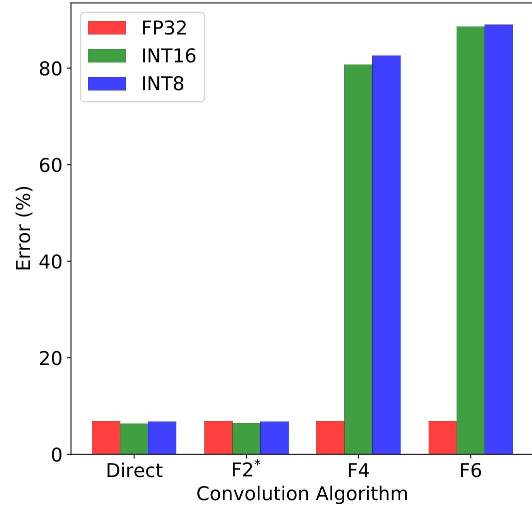


Figure 4.4: Replacing the convolutions in pre-trained ResNet-18 models on CIFAR-10. Winograd works well in FP32, but accuracy drops drastically with quantization for larger tiles. Quantized configurations performed a *warmup* of all the moving averages in Eq.4.1. Without this relaxation, requiring a Winograd-aware layers as in fig. 4.5, $F2$ would be unusable.

classification error of ResNet-18 for CIFAR-10 when trained at different bitwidths using standard convolutions and then replacing them for deployment. The numerical error is, to a large extent, the main limiting factor for adopting large-tiled Winograd and for adopting Winograd convolutions in general for reduced precision networks.

In this work we focus on minimizing the numerical errors that arise when using the Winograd algorithm in quantized networks. Our approach does not aggravate the compute and memory challenges previously mentioned. Instead, it indirectly alleviates these by making use of quantization.

4.4 Winograd-aware Networks

In this section we present a method to overcome the challenge of deploying fast and accurate CNNs making use of Winograd convolutions and 8-bit integer arithmetic. First we describe how we address the numerical error problem during training by means of Winograd-aware layers and, introduce a relaxation on how the transformation matrices G, $B^{\top}$ and $A^{\top}$ are used by including them to the set of learnable parameters. After that, in

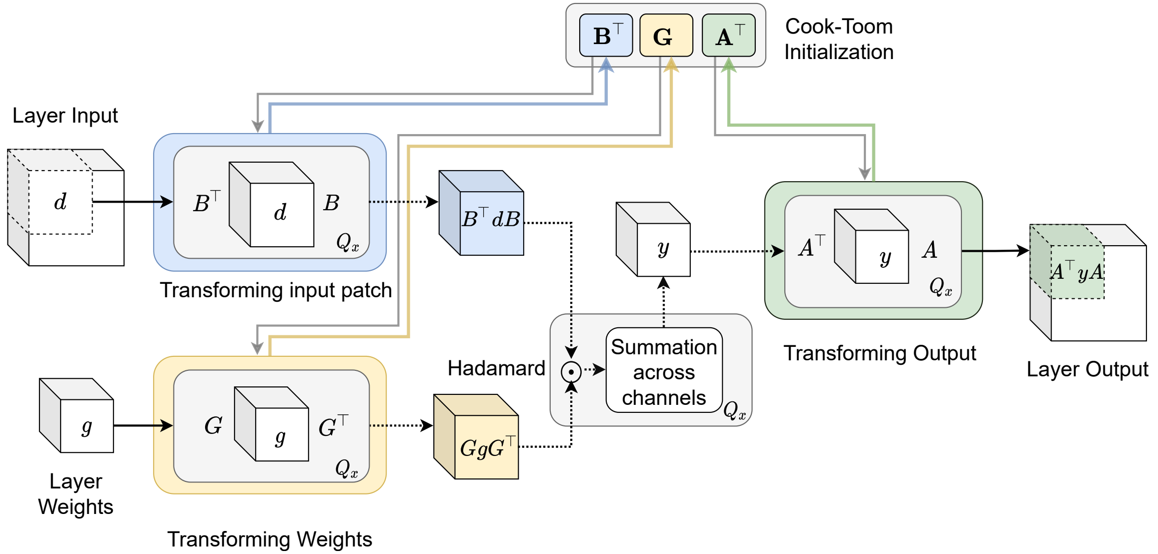


Figure 4.5: Forward pass of Winograd-aware layers. Transformation matrices G, $B^\top$ and $A^\top$ are constructed via Cook-Toom. If these are included in the set of model parameters, they would be updated with every batch via back-progation (this is represented with the coloured arrows going back to matrices G, $B^\top$ and $A^\top$, carrying the gradients to update the values of each transform). In its default configuration, each intermediate output throughout the pipeline quantized to the same level as the input and weights, this is represented by Q_x.

Section 4.4.2 we present a framework that, given a real latency model, it jointly optimizes a given macro-architecture for accuracy and latency leveraging Winograd-aware layers.

4.4.1 Winograd-aware Convolutional Layers

Neural networks have proven to be resilient to all kinds of approximations, e.g. pruning and quantization. When applying these techniques, consistently better models are generated if these approximations are present during training (as oppose to applying these as a post-training stage). In other words, when the training is *aware* of quantization, or when training is *aware* of pruning.

Following this intuition, we propose an end-to-end Winograd-aware pipeline as shown in Figure 4.5. In the forward pass we apply Equation (4.1) to each patch of the activations from the previous layer. We can apply standard back-propagation, since Equation (4.1) is only a collection of matrix-matrix multiplications. This implementation allows us to:

- **Learn better filters.** Building an explicit implementation of each of the stages involved in the Winograd transform exposes the numerical errors introduced in Equation (4.1) to the learning of the filters. This prevents the accuracy drops shown in Figure 4.4 by learning in the transformation space that is ultimately used for deployment. This formulation also acts as a regularization mechanism.

- **Learn the transforms.** Traditionally, matrices G, $B^\top$ and $A^\top$ are fixed after being initialized using the Cook-Toom algorithm and equal for all $F(m \times m, r \times r)$ in the model, given the same m and r. Instead, we can treat them as another set of *learnable* parameters in the layer. This relaxation leads to much improved performance in quantized networks while still maintaining the overall structure of the Winograd convolution algorithm and its speedups. Including these matrices as part of the model marginally increases the model size.

- **Quantization diversity.** Unlike standard convolutions, which does not require intermediate computation (i.e. the input and weights are directly convolved), Winograd convolution requires at least four of different stages, namely $GgG^\top$, $B^\top dB$, the Hadamard product and the output transformation. Each of these can be quantized to a different number of bits depending on the bitwidth of the input, that of the weights, and the overall complexity of the problem the network is designed to solve. This level of quantization granularity is possible with our Winograd-aware formulation.

4.4.2 Winograd-aware Neural Architecture Search

Simultaneously maximizing accuracy and minimizing latency with Winograd convolution is not trivial. The reason for this is that large tiles can result in low latency, but come at the cost of higher numerical error. This presents a good opportunity to jointly optimize network accuracy and latency.

To this end, we implement a NAS-based approach that automatically transforms an existing architecture into a Winograd-aware version. We perform NAS at the micro-architecture level by selecting from different convolution algorithms for each layer, but without modifying the network's macro-architecture (e.g. number or order of layers, number of channels in each layer, etc). Keeping the macro-architecture fixed allows us to fairly compare between the standard model, making use of algorithm such as `im2row`, to its Winograd-aware counterpart in terms of latency and accuracy. We call our framework `wiNAS`.

Introducing latency measurements into the optimization objective requires knowing the shape of the input tensor, i.e. the activations from the previous layer, at each layer of the network. We design `wiNAS` as a variation of ProxylessNAS [47], leveraging path sampling while performing the search. This technique, enables the allocation of the entire network on a single GPU by evaluating no more than two candidate operations at each layer per batch. Similarly to ProxylessNAS, `wiNAS` formulates the search as a two-stage process,

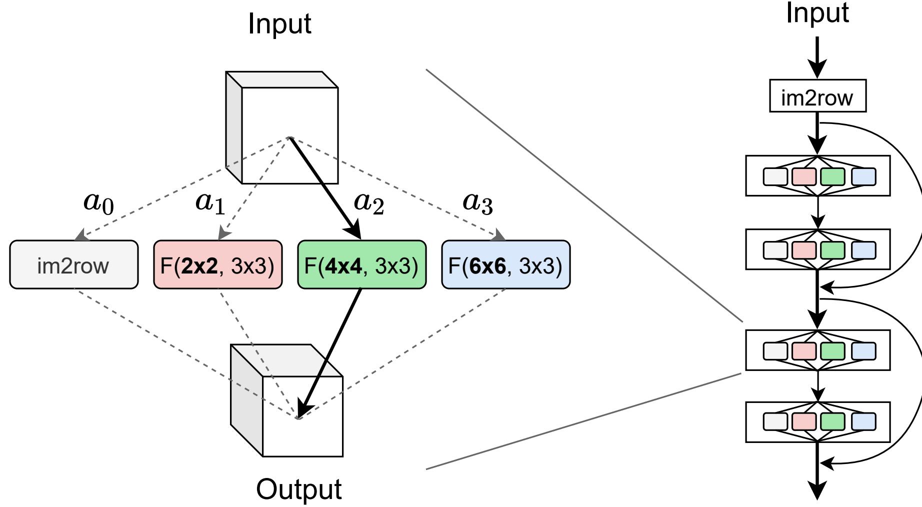


Figure 4.6: With `wiNAS`, each 3×3 convolution in a given architecture (on the left, the first 2 residual layers of a ResNet18 network are shown) is implemented with either `im2row` or with Winograd convolutions varying tile size. While the former is lossless and faster than direct convolution, Winograd offers lower latencies but introduce numerical instability that could ultimately impact in accuracy. The first convolutional layer of the macro-architecture is fixed to use standard convolutions since, due to the large input dimensions, Winograd convolutions do not offer speedups (as shown in Figure 4.12). `wiNAS` does not change the network's macro-architecture, it optimizes the choice of convolution algorithm to balance accuracy, latency reduction and, numerical error.

alternating the update of model parameters (the weights), where the loss is defined as:

$$L_{weights} = L_{CE} + \lambda_0 \left\|w\right\|_2^2 \tag{4.2}$$

where L_{CE} stands for the standard cross-entropy loss. To update of architecture parameters (the weight assigned to each operation on a given layer), the loss introduces the latency metrics term as:

$$L_{arch} = L_{CE} + \lambda_1 \left\|a\right\|_2^2 + \lambda_2 E[latency] \tag{4.3}$$

where a are the architecture parameters and λ_2 controls the impact of latency in the loss. The expected latency, $E\{latency\}$, for a given layer is the weighted combination of the latency estimate of each candidate operation with their respective probability of being sampled. Intuitively, searching for Winograd convolutions with high λ_2 would result in faster models, potentially at the detriment of accuracy.

Unlike ProxylessNAS, which performs the search at the macro-architecture level, `wiNAS` focuses on selecting the optimal convolution algorithm for each of the 3×3 convolutional layers. Therefore, the set of candidate operations for a given convolutional layer contains `im2row` and Winograd-aware layers in their $F2$, $F4$ and $F6$ configurations. This search

space is illustrated in Figure 4.6. Each candidate operation comes with its respective latency, which is a function of the output dimensions and quantization level.

4.5 Experimental Setup

In this section we describe each experiment we conducted. These are grouped into three categories: first we study the suitability of Winograd-aware layers to train better networks; then we perform a study on the speedups achievable by Winograd convolutions on mobile CPUs; finally, we measure the capabilities of `wiNAS` to optimize a given architecture to make it accurate and fast when running on off-the-shelf hardware. We used PyTorch [272] for training and Arm Compute Library to obtain latency measurements.

4.5.1 Vanilla Winograd-aware Networks

We begin our study of Winograd-aware networks by performing an extensive evaluation on the ResNet-18 [146] architecture using the CIFAR-10 [199] dataset. In this experiment we train the network end-to-end using standard convolutions, $F2$, $F4$ and $F6$ Winograd convolutions. For each experiment with Winograd, all layers in the network use the same tile size, except the last two residual blocks which are kept fixed to $F2$ since activations from the preceding layers are 4×4, the size of the input tile required for $F2$. The input convolutional layer uses normal convolutions. We run the experiments for FP32, INT16, INT10 and INT8 quantized networks, where both weights and activations are uniformly quantized (including all the intermediate outputs shown in Figure 4.5). We follow the per-layer symmetric quantization as described in Krishnamoorthi [198] except for the input layer and the output fully connected layer, which are always in full-precision. We repeated each experiment while enabling the Winograd transforms G, $B^\top$ and $A^\top$ to be learnt, which we denote using the additional suffix *-flex*. These transforms are always quantized to the same bitwidth as weights and activations are.

Winograd-aware layers do not require an over-parameterized model to perform well. We also varied the model size by using a width-multiplier, as used by the MobileNets family, ranging from 0.125 to 1.0, meaning that when the multiplier is 1.0 the network is the full ResNet-18. This leads to models ranging between 215K and 11M parameters. Winograd-aware layers with learnable transformations marginally increase (less than 0.1%) the model size, since the transforms themselves need to be saved for model deployment. We repeated the experiment for CIFAR-100 [199], but without varying the depth-multiplier.

CIFAR-100 is considerably more challenging that CIFAR-10, as it is comprised of 100 classes with only 600 images per class.

Additionally, we use an INT8 LeNet [210], trained on the MNIST dataset, to evaluate the suitability of Winograd-aware layers with learnable transforms for 5×5 filters. This is a more challenging case than 3×3 filters, because a larger tile tile is required, as defined by $F(m \times m, r \times r)$. With larger transformation matrices, choosing of more good polynomial points is also required.

For ResNet-18 models, we replace 2×2-stride convolution layers with a 2×2 max-pooling layer followed by a dense 3×3 convolution layer. Altering the network in this way is necessary since there is no known equivalent for strided Winograd convolutions, which remains an open research question. This is a common strategy when evaluating Winograd [232, 65]. We also halved the number of output channels of the input layer in order to reduce the memory peak during training. We use the Adam [191] optimizer, batch of 64 training examples, a 90:10 dataset split for train and validation and train for 120 epochs. Both CIFAR-10/100 use the same ResNet-18 architecture, differing only in the number of outputs of the fully connected layer. We constructed the transformation matrices as in [335]. Results for other popular CNN architectures are shown in Appendix A.6.

4.5.2 Evaluating Winograd-aware NAS

To evaluate `wiNAS`, we define two different sets of *candidate operations*. These spaces are: $\texttt{wiNAS}_{\texttt{WA}}$ and $\texttt{wiNAS}_{\texttt{WA-Q}}$, both allowing each 3×3 convolutional layer to be implemented with either `im2row` or each of the Winograd configurations, $F2$, $F4$ or $F6$. The former uses a fixed bitwidth for all elements in the architecture, while the latter introduces in the search space candidates of each operation quantized to FP32, INT16 and INT8.

The hyperparameters used for `wiNAS` are as follows: for the learning of model parameters we use mini-batch SGD with Nesterov momentum [314]. In the stage where we update the architecture parameters we use instead Adam with the first momentum scaling, β_1, set to zero, so the optimizer only updates paths that have been sampled in the current step. For both stages we use Cosine Annealing [236] scheduling and a batch size of 64. We perform the search for 100 epochs in each search space at different λ_2 values ranging from 0.1 to 1e-3. Once the search is completed, we trained the architecture end-to-end with the same hyperparameters as the rest of winograd-aware networks, as described earlier.

SoC	Core				RAM	GPU
	CPU	Clock	L1	L2		
Kirin 960	Cortex-A73	2.4 GHz	64 KB	2048 KB	3GB LPDDR4	Mali G71 MP8
	Cortex-A53	1.8 GHz	32 KB	512 KB		

Table 4.1: Key hardware specifications for the high-performance Cortex-A73 and the high-efficiency Cortex-A53 cores found on a HiKey 960 development board. Both are mobile CPUs widely available in off-the-shelf hardware.

4.5.3 Winograd Convolutions on Mobile CPUs

For our study, we chose Arm Cortex-A73 and Cortex-A53 cores running on a Huawei HiKey 960 development board with the big.LITTLE CPU architecture. These cores are good candidates for validating the speedups that are achievable with Winograd convolutions in today's off-the-shelf [4] mobile hardware.

While both A73 and A53 are implemented as 16nm quad-core CPUs, the former is a high-performance processor and the latter implements a high-efficiency processor. In Table 4.1 we summarize the main differences between these CPUs. The memory bandwidth and cache size would be the primary factors that ultimately sets the upper limit to the speedup achievable by Winograd since it requires operating in larger tiles than direct convolution algorithms such as `im2row` or `im2col`.

In our study, we measured the time taken for $3{\times}3$ convolutions using `im2row`, `im2col` and each of the Winograd configurations ($F2$, $F4$, $F6$) when varying output width/height (from $112{\times}112$ down to $2{\times}2$) and input/output channels, $inCh \rightarrow outCh$ (from $3 \rightarrow 32$ to $512 \rightarrow 512$). We performed the benchmark in controlled conditions and in single thread mode. Each combination was run five times with five seconds delay in between to prevent thermal throttling. We implemented Winograd convolutions using GEMMs as in Maji et al. [241], and performed the same experiment separately on the A73 and A53 for both FP32 and INT8. INT16 measurements were not supported by Arm Compute Library at the time this research was carried out. Therefore for experiments considering the $\texttt{wiNAS}_{\texttt{WA-Q}}$ space, latencies for INT16 were interpolated accordingly.

[4] These cores are also the building blocks of Qualcomm's Snapdragon 835 mobile platform, present in Samsung's Galaxy S8 and Google's Pixel 2, among others.

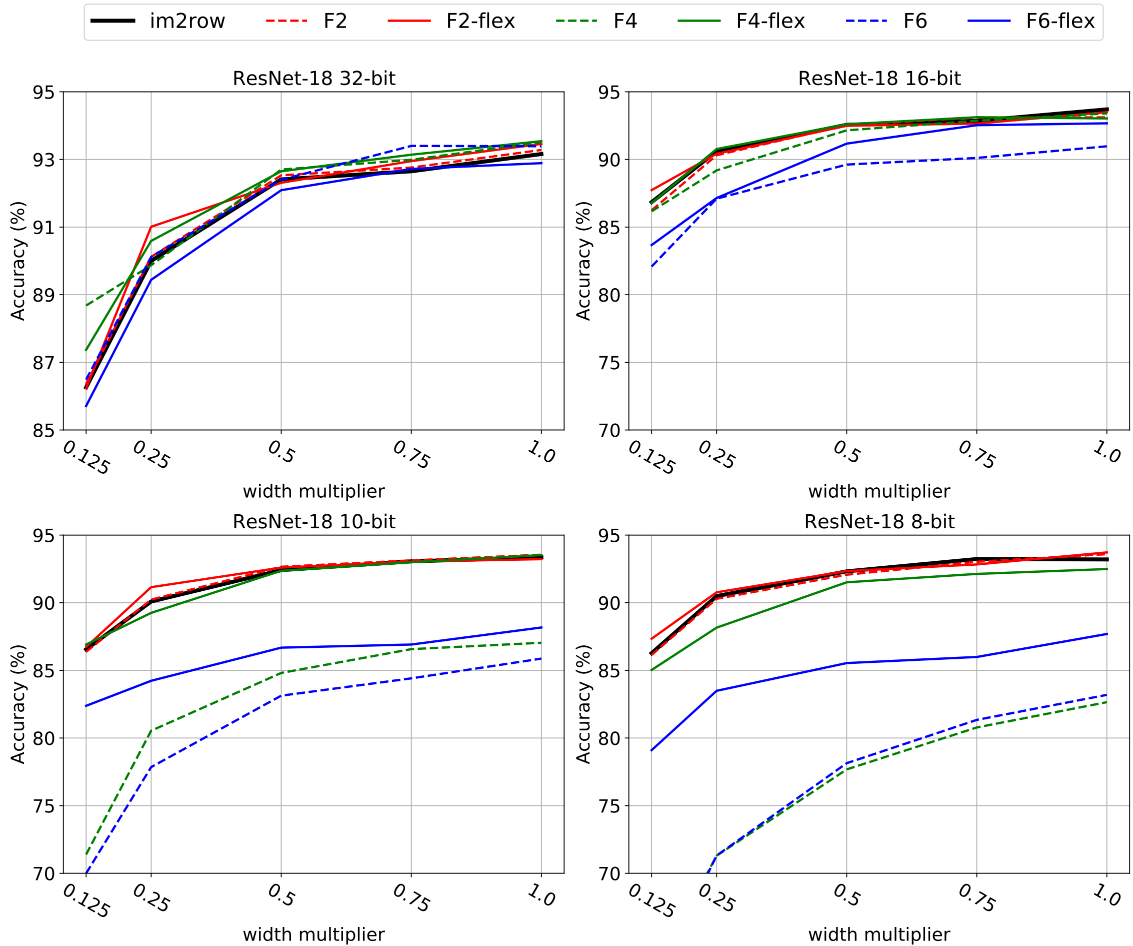


Figure 4.7: Performance of a Winograd-aware ResNet-18 at different bitwidths and trained with different Winograd configurations. We show how Winograd-aware layers scale with network's width. In quantized settings, models that learn the Winograd transforms (*-flex* configurations), strictly outperforms those models that keep them fixed with the values obtained via Cook-Toom.

4.6 Experimental Results

The results of this work are arranged as three subsections. First, we show that Winograd-aware networks can achieve high accuracy, even at INT8. Second, we discuss the results from our dense benchmark for Winograd convolutions on mobile CPUs. Third, we show that `wiNAS` can jointly optimize a given macro-architecture for accuracy and latency.

4.6.1 Winograd-aware Convolutions

Figure 4.7 (top left) shows Winograd-aware networks in FP32 perform as well as direct convolutions (represented under the `im2row` label), with both fixed and learned (*-flex*) transformation matrices. With quantization (all other plots), Winograd-aware layers are essential to enable the use of fast Winograd convolutions. This is not possible if switching

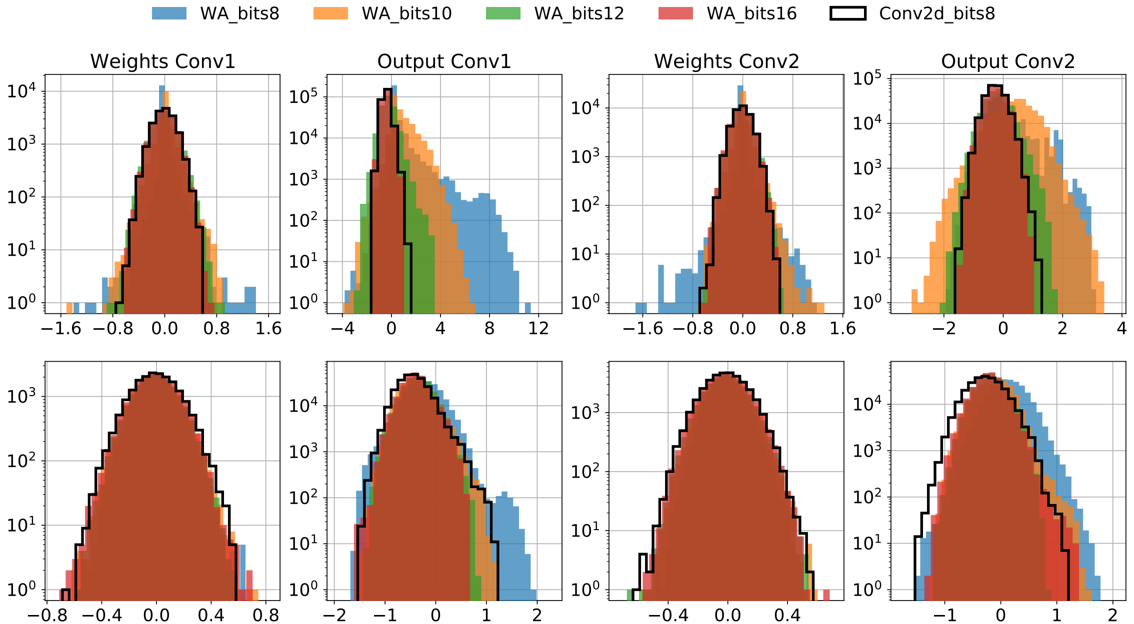


Figure 4.8: Distributions of weights and outputs of the $3{\times}3$ convolutional layers in the fourth residual block of a $F4$-ResNet18 network with width multiplier set to 0.25. Distributions for *static* winograd-aware layers are shown above for different bitwidths, below for *flex* layers. In black, the contour for the distributions from standard 8-bit convolutional layers. Distributions from *static* layers become wider as bitwidth is decreased for both weights and outputs. The latter is noticeable more distorted due to the errors introduced during the Winograd convolution with default transforms. This is not the case when these are learnt. The patterns shown in this figure appear across all residual blocks in the network.

to Winograd convolutions after training, as is commonly done in practice.

Furthermore, we show that learning the Winograd transforms (-*flex*) results in 10% and 5% better accuracies for $F4$ and $F6$ in INT8 scenarios. We argue that enabling this relaxation helps in easing the numerical instability inherent to Winograd convolutions, which is further exacerbated by quantization. The accuracy of Winograd-aware models scales linearly with network width, suggesting that these can be exploited in conjunction with architecture compression techniques such as channel pruning. In Figure 4.8 we compare side by side the distributions of weights and output tensors of Winograd-aware convolutional layers learning the transforms (bottom row) against those that keep them fixed (top row). While in the context of enough bitwidth, 16-bits as shown in the plot[5], distributions in both layers are very close to that of standard convolutions, shown its contour as a black line. However, as bitwidth is decreased, the distribution shift grows more evidently for *static* Winograd-aware layers. We further visualize in Figure 4.9 the impact on numerical degradation and the aforementioned distribution shift by projecting the inputs of each layer in

[5]This pattern is also observed at FP32 but we omit it to keep the visualization less cluttered.

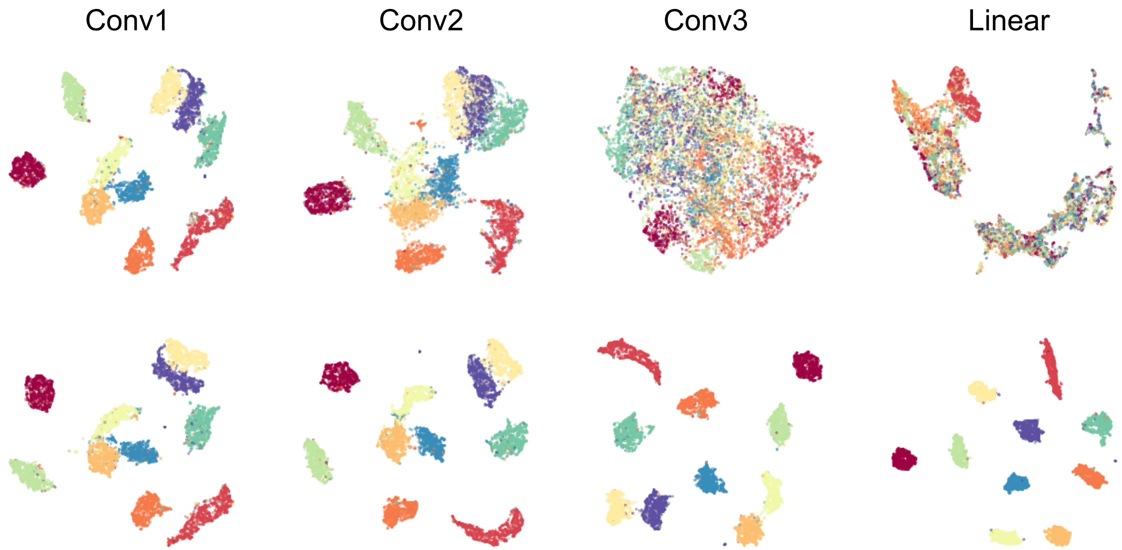


Figure 4.9: Clustering of inputs to each layer in a CNN for MNIST (10 classes) comprised of three $F(4 \times 4, 3 \times 3)$ layers followed by a linear layer. Both *flex* (below) and *static* (above) models were trained for 10 epochs and at 8-bits. The former reached 99.3% accuracy while the latter just 82.1%. After each *static* layer, the output tensor has degraded, resulting in overlapped clusters. This is not the case on *flex* layers. Once training is completed, we passed the test set (10K images) through the network and used a manifold approximation clustering mechanism [245] to project the input tensor at each layer. Each dot is a projection of a single input, there are 10K dots in each layer's projection.

a 3-layer CNN for MNIST. These results visually highlight the importance of learning the transforms showing that, when this is not the case, the numerical error accumulates rapidly, difficulting even the simplest image classification tasks.

Results from LeNet (5×5 filters), provides further evidence that larger tiles result in higher numerical error. In Figure 4.10, we show that even in relatively small datasets like MNIST, keeping the transformations G, $B^\top$ and $A^\top$ fixed, leads to poor results as the output tile size is increased. This difference is almost 47% in the case of $F(6\times6, 5\times5)$ layers, which uses 10×10 tiles.

Winograd-aware layers do not structurally modify the network architecture, since they only make use of different algorithm to perform convolution. We demonstrate it is possible to transform a pre-trained model with standard convolutions into its Winograd-aware counterpart within a few epochs. Concretely, in Figure 4.11 we show that an INT8 ResNet-18 $F4$ can be adapted from a model of the same network that was trained end-to-end with standard convolutions in 20 epochs of retraining. This represents a $2.8\times$ training time reduction for Winograd-aware models. This is only possible when allowing the transformation matrices to evolve during training. Adapting to Winograd-aware FP32 models can be done in a single epoch for both fixed and learned Winograd transformations.

We believe both $F4$ and $F6$ performance could be raised with alternative quantization implementations (e.g. by learning quantization ranges or using non-uniform approaches,

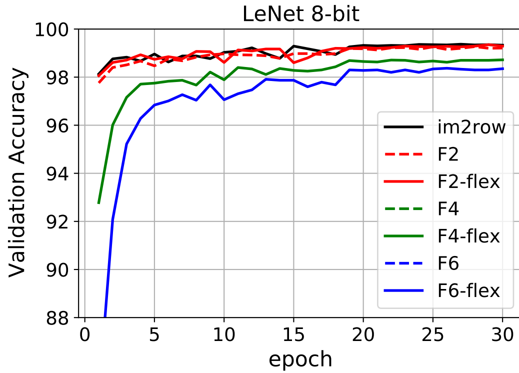


Figure 4.10: Performance of INT8 LeNet on MNIST using standard convolutions (`im2row`) or Winograd-aware layers. Letting the transformations to evolve during training (*-flex*) always results in better models. F4 and F6 configurations (not shown) reach an accuracy of 73% and 51%, respectively. All configurations reach 99.25% ± 0.1% in full precision.

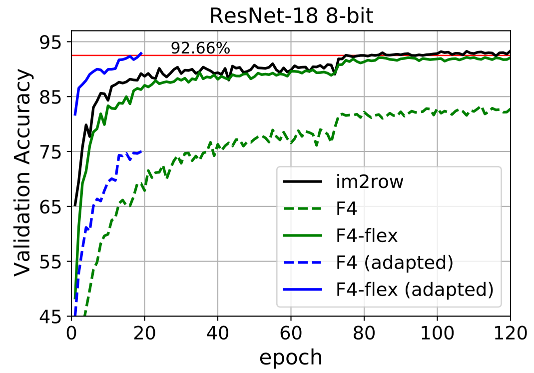


Figure 4.11: Transforming a standard model (trained with default convolutions) to its Winograd-aware counterpart can be done in very few epochs of retraining. We found this technique works best if the Winograd transformations are learnt during retraining (*-flex*), otherwise adaptation becomes much more challenging.

	inCh --> outCh																			
	3 --> 32				32 --> 64				128 --> 192				192 --> 256				256-->512			
outW	im2row	F2	F4	F6	im2row	F2	F4	F6	im2row	F2	F4	F6	im2row	F2	F4	F6	im2row	F2	F4	F6
2	0.007	0.008	0.016	0.029	0.070	0.043	0.082	0.167	0.659	0.407	1.219	2.196	1.463	1.082	2.378	4.407	3.912	2.932	6.619	11.853
4	0.011	0.029	0.016	0.030	0.154	0.078	0.081	0.167	1.642	0.802	1.170	2.195	2.884	1.731	2.502	4.486	7.450	4.962	6.588	11.947
6	0.021	0.053	0.065	0.029	0.328	0.199	0.174	0.165	4.137	2.229	2.040	2.148	6.780	4.559	4.135	4.327	17.450	13.858	11.452	11.919
8	0.031	0.059	0.064	0.133	0.519	0.280	0.175	0.408	5.306	2.993	2.004	3.899	10.932	6.145	4.167	7.907	28.238	14.930	11.499	21.241
10	0.058	0.101	0.119	0.144	0.910	0.475	0.482	0.412	9.466	5.054	5.321	3.973	17.808	10.198	10.318	7.904	44.656	27.597	32.685	21.437
12	0.066	0.133	0.129	0.132	1.208	0.621	0.475	0.424	11.625	6.601	5.382	3.971	24.196	12.995	10.272	7.955	61.236	35.702	32.164	21.478
14	0.087	0.186	0.154	0.267	1.610	0.868	0.695	1.043	16.177	9.277	7.498	9.846	33.702	18.154	14.220	19.082	85.809	48.590	34.306	60.003
16	0.111	0.235	0.153	0.283	2.592	1.191	0.723	1.051	20.845	12.158	7.551	10.002	42.362	23.147	14.310	19.263	109.943	57.083	34.190	60.504
18	0.169	0.281	0.263	0.281	3.315	1.379	1.133	1.031	26.785	15.125	12.159	9.961	55.085	29.292	23.178	19.476	142.460	75.505	63.799	60.987
20	0.184	0.325	0.249	0.400	3.416	1.695	1.131	1.728	32.851	18.450	12.115	15.108	67.300	35.276	23.274	27.723	173.488	90.041	65.349	67.923
22	0.210	0.398	0.331	0.410	4.164	2.070	1.506	1.690	40.245	22.207	16.010	15.114	82.028	43.166	30.697	27.781	213.326	110.160	82.434	67.228
24	0.247	0.452	0.324	0.409	4.783	2.453	1.498	1.729	47.961	26.600	16.126	15.035	97.706	51.064	30.954	27.923	251.771	125.604	83.167	67.047

Figure 4.12: Latencies (in milliseconds) of convolving increasingly larger input tensors in the width/height dimensions (y axis) and in depth (x axis). We compare the time needed for `im2row` and each of the Winograd configurations with 32-bit arithmetic on a Cortex-A73. We show that (1) `im2row` is the consistently the optimal algorithm for the input layer to a network, (2) the choice between $F2$, $F4$ and $F6$ should be done based on the output's width/height and, (3) this choice should not generally be altered based on $inCh \rightarrow outCh$.

as discussed in Section 2.1.2.1), closing the accuracy gap with $F2$ and direct convolutions.

4.6.2 Benchmarking Winograd Convolutions on Mobile CPUs

The speedups associated to with the use of Winograd convolutions often only account for the point-wise stage while assuming negligible costs for the input, weights and output transformations. Furthermore, these also assume that the larger the input patch, *d*, the larger the speedup compared to normal convolutions. However, although these assumptions are true

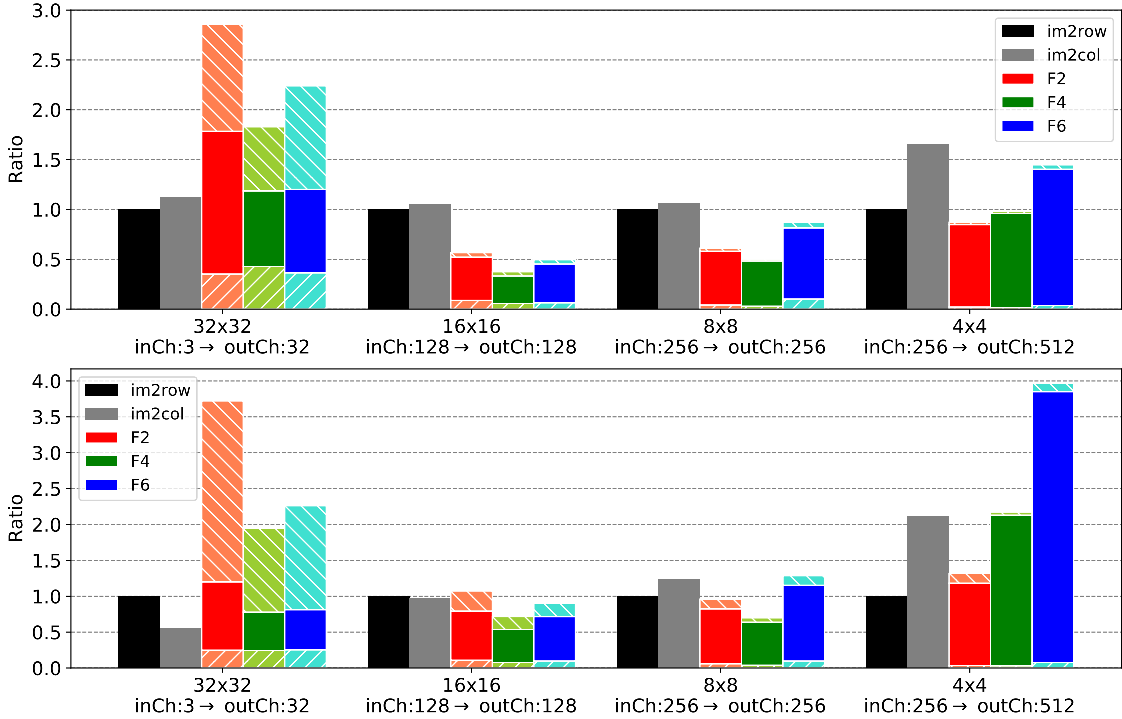


Figure 4.13: Latencies (normalized *w.r.t* `im2row`) to execute different layers of the ResNet-18 network measured in A73 (above) and A53 (below). For Winograds, solid colour bar regions represent the element-wise Hadamard GEMM stage, below and above it are respectively the input and output transformation costs. Measured for FP32 and with standard Winograd transforms.

for large tensors, they are not necessarily true when working with tensors of the sizes often found in CNNs for image classification or object detection.

Figure 4.12 shows a small portion of the obtained latency measurements for our benchmark in FP32. An almost identical pattern appears when using 8-bit arithmetic. In Figure 4.13 we show the speedups that Winograd convolutions offer at different layers of a ResNet-18 network. Our observations can be summarized in three points:

Input layers do not benefit from Winograd. This is primarily because the matrices in the element-wise GEMM stage are not large enough to compensate for the costs of transforming the input and output patches (see the leftmost part in Figure 4.12 and Figure 4.13). They represent a significant portion (up to 65% and 75% respectively on the A73 and A53) of the total costs for convolving an RGB 32×32 input expanded to 32 channels. Similar ratios can be observed for other input sizes. In spite of this, this first layer accounts for a marginal portion of the total latency of the model, often below 1ms for these platforms.

Optimal m is a function of input width and height. For an input with sufficient number of channels, e.g. 32 channels and above, we observe in Figure 4.12 a consistent pattern alternating between $F4$ and $F6$ as the channel dimension of the output increase.

This alternation comes as a result of the impossibility of subdividing the input into an integer number of $(m + r - 1) \times (m + r - 1)$ patches, and therefore having to waste calculations when operating around the matrix edges. This pattern is invariant to different $inCh \rightarrow outCh$ configurations and fades away as input dimensions exceed 40×40, where $F6$ consistently becomes the fastest (not shown).

Winograd transforms are costly. Excluding inputs with very few channels, the cost of performing the transformations to and from the Winograd domain can exceed 25% of the overall costs. These costs become negligible as the input width and height decrease, but the rate at which this happens also depends on the hardware. Our Winograd-aware pipeline formulation does not impose any constrains on how the transforms are learnt. This results in dense transforms (as opposed to the default transforms which sometimes contain zeros) and therefore applying them might require additional compute. Table 4.2 includes this latency overhead in models making use of learned transforms. In Appendix A.7 we provide more details on how dense transforms impact overall latency.

As shown in Table 4.2, the speedups from FP32 Winograd convolutions are smaller on the A53 than on the A73. We argue this comes as a results of the differences in the memory subsystem (Table 4.1), limiting the lower-end CPU to efficiently operate with larger tensors. These speedups grow significantly when leveraging INT8 arithmetic, made possible by Winograd-aware training. Concretely, INT8 Winograd increases the speedup on the A53 by a factor of almost $1.5\times$ compared to Winograd in FP32, as shown in WA_{F4} configurations – at the cost of 0.7% accuracy in CIFAR-10. In the case of the more challenging CIFAR-100 dataset, the drop in accuracy is more severe. However, our WA_{F2} layers offer attractive speedups for INT8 with no drop in accuracy. We rely on `wiNAS` to minimize this degradation with small impact on latency.

4.6.3 Joint micro-architecture Optimization with `wiNAS`

Choosing the convolution algorithm that minimizes overall network latency can be easily done by looking at the latency benchmark results. However, since the accuracy of Winograd convolutions degrades rapidly in reduced precision networks, selecting the fastest algorithm for each layer without sacrificing accuracy is not straight forward.

When using `wiNAS`, values of λ_2 larger than 0.05 consistently result in models with the same layer configuration as those in WA_{F4} (described in Section 4.5.1). When lowering the impact of latency in Equation (4.3) loss function, we observed several $F4$ Winograd-aware layers were replaced with either `im2row` or $F2$, at the cost of less than 9 ms latency increase in the worst case, for an INT8 model on the A53 for CIFAR-100. These models resulted in similar accuracies in FP32 and reached 0.32% and 1.1% higher accuracies in

Conv. Type	Bit-width	Accuracy (%) CIFAR-10	Accuracy (%) CIFAR-100	Cortex-A53 Latency	Cortex-A53 Speedup	Cotex-A73 Latency	Cotex-A73 Speedup
`im2row`		93.16	74.62	118	-	85	-
`im2col`		93.16	74.62	156	0.76×	102	0.83×
W_{F2}		93.16	74.60	126	0.94×	56	1.52×
W_{F4}	32 / 32	93.14	74.53	97	1.22×	46	1.85×
WA_{F2}*		93.46	74.69	126	0.94×	56	1.52×
WA_{F4}		93.54	74.98	$122^{\dagger}$	0.92×	$54^{\dagger}$	1.58×
$\texttt{wiNAS}_{\texttt{WA}}$		93.35	74.71	$123^{\dagger}$	0.96×	$56^{\dagger}$	1.52×
`im2row`		93.20	74.11	117	1.01×	54	1.57×
`im2col`		93.20	74.11	124	0.95×	59	1.45×
WA_{F2}*	8 / 8	93.72	73.71	91	1.30×	38	2.24×
WA_{F4}		92.46	72.38	$82^{\dagger}$	**1.44×**	$35^{\dagger}$	**2.43×**
$\texttt{wiNAS}_{\texttt{WA}}$		92.71	73.42	$88^{\dagger}$ / $91^{\dagger}$	1.34× / 1.30×	$35^{\dagger}$ / $36^{\dagger}$	2.43× / 2.36×
$\texttt{wiNAS}_{\texttt{Q}}$	auto	92.89	73.88	$74^{\dagger}$ / $97^{\dagger}$	**1.60×** / 1.22×	$32^{\dagger}$ / $43^{\dagger}$	**2.66×** / 1.98×

Table 4.2: Performance in terms of accuracy and latency (ms) of ResNet-18 when convolutions are implemented with different algorithms and for different quantization levels. We show that Winograd-aware ($\text{WA}_{\text{F2/4}}$) layers combine the speedups of Winograd convolutions with those of INT8 arithmetic, with little to no accuracy loss in some cases. This is not possible with existing Winograd ($\text{W}_{\text{F2/4}}$) formulations. Latency is measured on Arm Cortex-A73 and A53 cores. For the last two rows, `wiNAS` found different optimizations for each dataset. We show latencies for CIFAR-10 on the left and CIFAR-100 on the right. Speedups are shown against `im2row` in FP32. (*) With default Winograd transforms. (†) Includes worst case latency increase due to be using learned transform, which are often dense.

INT8 for CIFAR-10 and CIFAR-100 respectively. Despite CIFAR-100 models sacrificing more latency in order to recover accuracy, they remained faster than WA_{F2} at INT8.

When introducing quantization into the search, as in $\texttt{wiNAS}_{\texttt{WA-Q}}$, the accuracy gap is almost closed for CIFAR-10 and further reduced for CIFAR-100. This comes primarily as a result of relying on higher bitwidths for the first layers in the network. In both cases, we maintain attractive speedups compared to `im2row` and Winograd convolutions in FP32, specially on the A73. These resulting architectures are described in Appendix A.8.

4.7 Discussion

In this section we present some of the challenges of training Winograd-aware networks and propose lines of future work.

High memory peak during training. A direct implementation of Equation (4.1) requires saving the intermediate outputs of each matrix-matrix multiplication, since these are needed

for back-propagation. This results in high memory usage. In this work we had to rely on *gradient checkpointing* [58] to lower the memory peak during training, at the cost of additional computation. We believe a native CUDA implementation of the Winograd-aware layers with better memory reuse would ease this problem. This is the main limitation we faced in order to scale the networks to ImageNet which results in much larger activation tensors and therefore higher memory overheads for Winograd-aware layers.

Using larger tiles. Learning larger models (with width multipliers 0.75 and 1.0) proved challenging for $F4$ and $F6$ when introducing quantization, as these configurations introduce more numerical error. Using other types of quantization, such as per-channel affine quantization as in Jacob et al. [175], would likely help. In addition, enabling different bitwidths throughout Equation (4.1) could help mitigate the accuracy drop by first identifying which elements are more sensible to reducing bitwidth. This level of quantization granularity is not considered in the evaluation in Section 4.6 but is discussed in the formulation presented in Section 4.4.1. The analysis shown in Figure 4.12 indicates that $F6$ Winograd-aware layers would offer substantial speedups in some particular configurations compared to $F4$. This raises the following question: *can we extend `wiNAS` and allow for small changes in the model macro-architecture, in particular introduce average pooling layers that reduce the spatial dimensions of activations without incurring into accuracy degradation?* This would enable for example replacing a $192 \rightarrow 256$ layer that generates $14{\times}14$ activations where F4 Winograd convolutions require 14.2ms on a Cortex-A73 into a layer that outputs a $12{\times}12$ tensor. This can be implemented with $F6$ Winograd convolutions in 7.9ms, resulting in a $1.8\times$ speedup.

Data-driven polynomial points discovery. As it was first discussed in Section 4.2.2, poorly chosen polynomial points when constructing G, $B^\top$ and $A^\top$ introduce significant deviations in the result of computing Winograd convolutions compared to that of normal convolutions. We observed that good starting points are also important even when learning the Winograd transformations, as Winograd-aware layers do. Polynomial points specifically tailored for quantized Winograd could alleviate some of the degradation that we observed with increased tile size.

Implications of learning Cook-Toom transforms. The formulation presented in Section 4.4 freely lets the transformation matrices change during training. As a result, these turned to be, unlike the default Cook-Toom transforms, dense in general. This resulted in an 20% impact in latency for $F4$ layers, limiting the speedups attainable with our method.

In spite of this, or approach still largely outperforms INT8 `im2` approaches. We elaborate on this point in Appendix A.7. Another implication of learning the transforms is that the operation performed in Winograd-aware layers is not a convolution, despite it sharing all the elements, speedups, overheads and, stages of the standard Winograd algorithm for convolutions. This naturally raises the question of *what is this convolutional-like operation actually doing?*. A preliminary analysis is briefly discussed in Appendix A.9. Having further insights would also opens up the possibility for further optimizations in how Equation (4.1) is implemented such as fusing some of the stages in the Winograd convolution as a first step towards designing a Hadamard-only convolutional-like layer. This would allow getting rid off some of the overheads discussed in Section 4.6.2.

4.8 Summary

Running CNN-based applications that require standard convolutional layers (i.e. not depthwise convolutions) is challenging in compute-constrained devices such as mobile CPUs. This chapter considered the old standing problem of numerical precision in Winograd conovlutions that has prevented their usage when deploying quantized models. We presented Winograd-aware layers as the building block to combine the benefits of quantized networks and fast Winograd convolutions while significantly alleviating the numerical error inherent to this algorithm. We studied Winograd-aware layers with different tile sizes, three quantization levels and on three popular datasets. We found that allowing the transformation matrices to evolve during training resulted in significantly better models. With `wiNAS` we leveraged Winograd-aware layers and latency metrics from off-the-shelf mobile CPUs and found architectures that helped minimize the numerical instability of Winograd. A Winograd-aware ResNet-18 quantized to INT8 offers up to $1.32\times$ faster inference for only a marginal accuracy drop compared to existing Winograd implementations, which are limited to FP32. This network is also $1.54\times$ faster than an optimized `im2row` implementation using INT8 arithmetic on a Cortex-A73 mobile CPU.

Chapter 5

Topology-aware Quantization of Graph Neural Networks

In Chapter 3 and Chapter 4, this thesis presented two orthogonal methods to accelerate inference on CNNs, a type of architecture that has been applied to multiple domains including vision, text and, speech. This chapter draws attention to a newer domain of ML that operates on graph-structured data, with models following a message-passing formulation and, that excel at a wider range of, often very different, types of applications.

Graph Neural Networks (GNNs) have received substantial attention in recent years due to their ability to model irregularly structured data. As a result, they are extensively used for applications as diverse as molecular interactions [94, 374], social networks [137], recommendation systems [326, 419] or program understanding [12, 396, 265]. Recent advancements have centered around building more sophisticated models [259, 137, 331, 352], including new types of layers [192, 329, 381, 317, 40] and better aggregation functions [70]. However, despite GNNs having few model parameters, the compute required for each application remains tightly coupled to the input graph size. For examples, a 2-layer Graph Convolutional Network (GCN) model with 32 hidden units would result in a model size of just 81KB but requires 19 GigaOPs to process the entire Reddit graph. That is roughly $5\times$ more than the number of operations needed to perform inference with an ImageNet-scale ResNet-18, despite this one having $140\times$ more model parameters. We illustrate this growth in Figure 5.1 and compare it with other popular models for image classification.

One major challenge with graph architectures is therefore performing inference efficiently, which limits the applications they can be deployed for. For example, GNNs have been combined with CNNs for SLAM feature matching [294], however it is not trivial to deploy this technique on smartphones, or even smaller devices, whose neural network accelerators often do not implement floating point arithmetic, and instead favour more efficient integer arithmetic. Integer quantization is one way to lower the compute, memory and

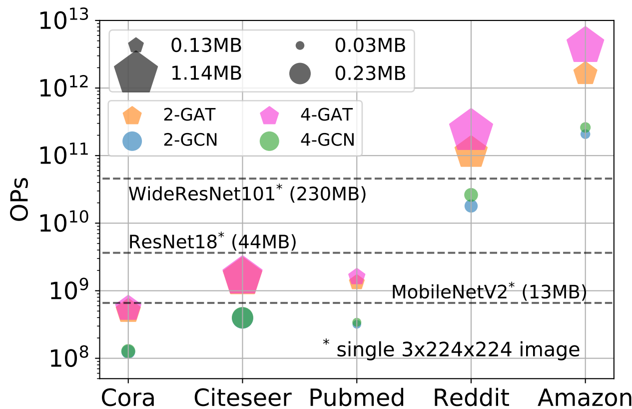


Figure 5.1: Despite GNN model sizes rarely exceeding 1MB, the OPs needed for inference grows at least linearly with the size of the dataset and node features. GNNs with models sizes 100× smaller than popular CNNs require many more OPs to process large graphs.

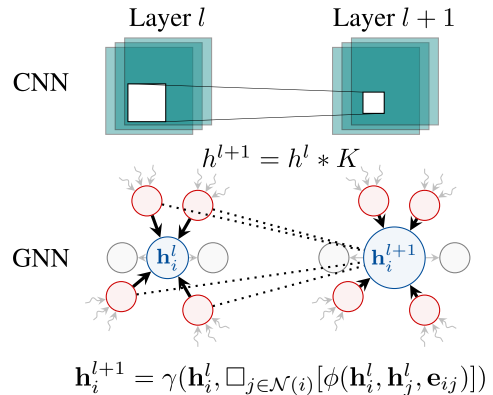


Figure 5.2: While CNNs operate on regular grids, GNNs operate on graphs with varying topology. A node's neighborhood size and ordering varies for GNNs. Both architectures use weight sharing.

energy budget needed to perform inference, without requiring modifications to the model architecture. In addition, quantization is also useful for model serving in data centers operating on, for example, very large social networks of product recommendation graphs.

Research Question: *What are the unique challenges that arise when quantizing graph neural networks? How do the different types of layers found in the literature perform under reduced bitwidth given the same input graph?*

5.1 Problem Setting and Contributions

There is no doubt that quantization has been well studied for CNNs and language models [175, 347, 402, 274]. However, before the contents of this chapter were first first published [316], there was no work explicitly characterising the issues that arise when quantizing GNNs or showing latency benefits of using low-precision arithmetic. Instead, works addressing GNN efficiency focused on other orthogonal techniques which include the use of knowledge distillation to support deeper but compact models [385], adding better support for distributed processing of GNNs [180], sampling methods [138, 405] and, hardware accelerators [262, 404, 59].

If we take a GCN layer as an examples, at a superficial level, quantizing a GCN and a standard convolutional layer from a CNN is a very similar process. Both require layer

parameters and inputs to be quantized in order to accelerate layer inference. Convolutional layers can be seen as a single-stage process, by which pre-quantized inputs and weights are convolved together to *directly* obtain the layer's output. In contrast, inference in GCN layers is performed as a three-stage process (as it was first described in Section 2.5), with each stage generating intermediate tensors. Therefore, each of these need to be quantized independently to ensure the entire *message-passing* pipeline can be accelerated using integer-only arithmetic. As it will be discussed in greater detail in Section 5.3, this is far from trivial as each stage varies on how susceptible it is to quantization degradation depending on the type of layer architecture, graph and, node in-degree.

This chapter considers the motivations and problems that arise when quantizing GNNs architectures. First, in Section 5.3 we provide an explanation of the sources of degradation in GNNs when using lower precision arithmetic. This analysis considers three popular GNN layers commonly found in the literature, namely GCN [192], GAT [329] and, GIN [381]. These layers are the building blocks for more sophisticated architectures. The results from this analysis show how the choice of straight-through estimator (STE) implementation, node degree, and method for tracking quantization statistics significantly impacts model quality. Then in Section 5.4, we propose `Degree-Quant` (DQ), an *architecture-agnostic* re-formulation of quantization-aware training for graphs that helps to overcome the problems identified in the previous section. With `Degree-Quant`, INT8 models often perform as well as their FP32 counterparts. At INT4, models trained with DQ typically outperform quantized baselines by over 20%. Unlike previous work, models trained with DQ generalize to *unseen graphs* without requiring modifications to their aggregation functions. We validate our study and proposed framework on six datasets for node classification, graph classification and regression for social graphs. We show that quantized networks achieve up to 4.7$\times$ speedups on CPU with INT8 arithmetic, relative to full precision floating point, with 4-8$\times$ reductions in runtime memory usage.

5.2 Background and Related Work

Many popular GNN architectures may be viewed as generalizations of CNN architectures to an irregular domain: at a high level, graph architectures attempt to build representations based on a node's neighborhood. However, unlike CNNs which operate on regular grids (e.g. images), a node's neighborhood does not have a fixed ordering or size. Because GNNs operate on irregular topologies, a number of challenges arise when attempting to accelerate these workloads. In this section we first provide a summary on how GNNs work

and present the main challenges that arise when doing inference. We then describe in detail other works that made use of quantization to compress and accelerate GNNs.

5.2.1 Graph Neural Networks: Overview and Challenges

Graph Convolutional Neural Networks (GCNs) can be interpreted as a generalization of standard convolutional layers to irregular grids (Figure 5.2). Inference involves a three-stage process by which: first, messages are generated by transforming the node features using the layer parameters; in the second stage, each node gathers and aggregates the messages of neighbouring nodes, often implementing a normalization mechanism; finally, a non-linearity is applied to the aggregated features at each node and these become the node features for the next layer. The specific implementation of these stages, *message generation*, *message aggregation* and, *feature update*, varies across layer architectures. Several of these layers were introduced in Section 2.5 including the notation used in this chapter.

Among the set of challenges that arise when processing GNNs two stand out by their impact in moving the field forward and by the increasing amount of approaches in the literature attempting to alleviate them:

Alternate execution patterns. Processing GNNs includes two very distinct types of workloads: first, a node-level workload where node features are multiplied by the layer weights (a dense `matmul`) as a first step to constructing the node messages; and, a graph-level workload where nodes gather the messages from neighbouring nodes during message aggregation. The former is similar to those workloads found in CNNs or language models which can benefit from high levels of data reuse. However, the latter often becomes memory bound due to many scattered memory accesses which result in very high rates of cache misses, severely slowing down inference and increasing energy consumption. Graph reordering can be useful to increase data reusability and better mapping to the system's cache hierarchy. However, a compromise between reordering quality and amortisation cost is unavoidable [26]. Regardless of the approach chosen, algorithms optimizing for maximum community overlap remain NP-hard [362].

Large activations. When graphs contain millions of nodes, training becomes challenging even on CPU-only settings with large amounts of RAM. The problem gets exacerbated as the community starts embracing deeper GNNs [180], requiring the saving of more, large activation tensors. To alleviate the memory burden of large graphs, sampling techniques have proven to be indispensable. Sampling also speeds up training and can act as a powerful regularizer [289]. However, graph sampling techniques have been evaluated for very

few layer architectures, mostly GCN, for multi-graph settings such as those for graph classification or graph regression applications. The impact of graph sampling on graphs with multi-dimensional edges features carrying rich semantics, such as source or intermediate representations of code, is yet to be understood.

We refer the reader to Section 2.5 for a more thorough introduction to GNNs as well an extended discussion on the challenges to accelerate them.

5.2.2 Reduced precision arithmetic for Graph Neural Networks

Orthogonal to graph sampling and reordering techniques, the use of low precision arithmetic can also contribute to accelerating inference on graphs. Unlike the previously mentioned techniques, quantization can be applied to all types of graph settings and, alleviates both the compute-bound and memory-bound workloads present in modern GNNs. Because of these reasons, the research community is steering towards adapting well studies quantization approaches for CNNs or language models to GNNs. The remaining of this section introduces works presented around the time and after ours was first published.

The work of Wang et al. [346] aims to accelerate the compute-bound node-level workload of projecting the node features to a new embedding space using the layer parameters. To this end, the authors learn a binary representation of the model parameters as well as a mapping of the inputs of each layer to a binary form. Bi-GCN makes uses of bucketing [10] in order to split the input and weight matrices into buckets, which are pairs of binary vectors with their corresponding scaling factors[1] (one per vector). This approach is the natural extension of XNOR-Net [281] to the graph setting in the sense that, instead of learning a scalar per sub-tensor (i.e. the input patch that overlaps with the filter in a CNN), a scalar is learnt per-node. Similarly to XNOR-Net, Bi-GCN also learns a scalar per output channel in the weight matrix. In this way, the dot product between input and layer weights can be accelerated using XNOR and bit count operations, in addition to, albeit comparatively very few, floating point operations to apply the scaling factors. With this, authors claim up to $47\times$, $41\times$ and $11\times$ speedup on Cora, PubMed and Reddit, respectively. However, it is unclear from the authors' analysis what the overall speedup is since inference on large graphs is known to be memory bound by irregular memory accesses during the aggregation stage and the workload that Bi-GCN accelerates often accounts for a small portion in the total GNN pipeline. It is also unclear if the transformed node features are binarized before performing aggregations, which would lead to a much higher memory usage than its full

[1]The reader should note that these scaling factors are not learnt directly but are instead defined at runtime based on the layer's weights and its inputs. We refer the reader to Section 2.1.2 for further details.

binary GNN counterpart. The $\sim 30\times$ model size compression claims are undeniable and becomes an important step towards deploying GNNs on more memory constrained devices.

The work of Bahri et al. [25] builds upon the changes introduced in XNOR-Net++ [45] over XNOR-Net, namely the introduction of learnable scaling tensors as opposed of using runtime-defined scalars, and adapt their usage to the graph setting. The training is framed as a cascaded distillation process where the network alternates its role as a full-precision teacher and a binarized student. To improve the convergence rate of such distillation process, a logit matching loss [157] is introduced. In this work, the authors expand the use of binary GNNs to point cloud graphs making use of EdgeConv layers by introducing: local structure preserving (LSP) [392] modules throughout the GNN guiding the student network to learn similar topological embeddings as those obtained during the preceding teacher training stage; and, a mechanism to accelerate the k-NN search of neighbouring nodes using binary node features and the Hamming distance as opposed to standard Euclidean distance with real-valued node features. Their evaluation on the ModelNet40 dataset [375] evidences that performing k-NN using binary node features remains unsolved (showing over 10% accuracy drop) unless using full precision weights. This suggest that acceleration from binary operations is limited to either the node-level feature extraction stage (as Bi-GCN does) or the k-NN search stage. Since the latter is found to dominate the overall run time for GNNs consuming point cloud data, an interesting approach would be to use integer arithmetic (e.g. 8-bits or 4-bits) to accelerate the node-level stage and, use a binarized representation of the projected node features to perform k-NN.

Also related to the work in this chapter is that of Feng et al. [101], where a heterogeneous quantization framework assigns different bits to the embedding and attention coefficients in each layer while maintaining the weights at full precision (FP32). Their framework sheds light on what elements in GNNs are more sensible to quantization and, show asymmetric quantization approaches with lower average bitwidth perform better than uniformly quantized GNNs with higher bitwidth. However, due to the mismatch in operands' bitwidth the majority of the operations are performed at FP32 after an unavoidable data casting stage, effectively vanishing speedups from integer arithmetic and, making it impractical to use in general purpose hardware such as CPUs or GPUs. In addition, they do not demonstrate how to train networks which generalize to *unseen* input graphs since their evaluation is limited to single-graph node classification datasets such as Cora and Reddit.

In contrast, `Degree-Quant` relies upon uniform quantization applied to all elements in the network and uses bitwidths (8-bit and 4-bit) that are supported by off-the-shelf hardware such as CPUs and GPU for which efficient low-level operators for common operations found in GNNs already exist. In addition, this chapter also considers the use of

`Degree-Quant` for graph classification and graph regression applications, expanding the use of quantizatized GNNs to unseen graphs.

5.3 Identifying the Sources of Error in GNNs

In this section, we build an intuition for why GNNs would fail with low precision arithmetic by identifying the sources of error that will disproportionately affect the accuracy of a low precision model. In particular, our analysis focuses on three models: GCN, GAT and GIN. This choice was made as we believe that these are among the most popular graph architectures, with strong performance on a variety of tasks [95], while also being representative of different trends in the literature.

As it was first introduced in 2.1.2.4, quantization-aware training (QAT) relies upon the straight-through estimator (STE) to make an estimate of the gradient despite the non-differentiable rounding operation in the forward pass. For integer QAT, the quantization of a tensor $\mathbf{x}$ during the forward pass is often implemented as:

$$\mathbf{x}_q = \texttt{clamp}\big(\lfloor \mathbf{x}/s \rceil + z, q_{\min}, q_{\max}\big) = \min(q_{\max}, \max(q_{\min}, \lfloor \mathbf{x}/s + z \rceil)) \tag{5.1}$$

where $q_{\min}$ and $q_{\max}$ are the minimum and maximum representable values at a given bitwidth and signedness, s is the scaling factor making $\mathbf{x}$ span the $[q_{\min}, q_{\max}]$ range and, z is the *zero-point*, which allows for the real value 0 to be representable in x_q. These are all statistics obtained at training time. Intuitively for a fixed number of bits, the wider the quantization range $[q_{\min}, q_{\max}]$ is, the less numerical resolution budget would be available to represent the values in $\mathbf{x}$.

In GNN layers, we identify the aggregation phase, where nodes combine messages from a varying number of neighbours in a permutation-invariant fashion, as a source of substantial numerical error. Outputs from aggregation have magnitudes that vary significantly depending on a node's degree: as it increases, the variance of aggregation values will increase. This asymmetry in the range of values observed makes the choice of a single set of $q_{\min}$ and $q_{\max}$ challenging to accurately represent the resulting values for all nodes. In addition, over the course of training $q_{\min}$ and $q_{\max}$ might become severely distorted by infrequent outliers, further reducing the resolution for the vast majority of values observed. Controlling $q_{\min}$ and $q_{\max}$ hence becomes a trade-off balancing the impact of outliers while preserving aggregation outputs from high degree nodes (i.e. *truncation error*) and, reducing the quantization ranges to increase numerical resolution (i.e. *rounding error*).

We can derive how the mean and variance of the aggregation output values vary as node degree increases for each of the three GNN layers. Making use of the formulation for MPNNs as in Section 2.5, we represent the output of the aggregation stage for node v_i whose degree is n as

$$\mathbf{o}_i^{(n)} = \mathbf{y}_i + \square_{j \in \mathcal{N}(i)}(\mathbf{y}_j) \tag{5.2}$$

, where $\mathbf{y_i}$ is the message from the i-th node, and, $\square$ a generic aggregation mechanism. For GIN, GCN, and GAT layers, and ignoring self-loops and layer indexing in the notation, such output vector is given by:

GIN	GCN	GAT
$\mathbf{o}_i^{(n)} = \sum_{j=1}^{n} \mathbf{y}_j$	$\mathbf{o}_i^{(n)} = \frac{1}{\sqrt{n}} \sum_{j=1}^{n} \frac{1}{\sqrt{d_j}} \mathbf{y}_j$	$\mathbf{o}_i^{(n)} = \sum_{j=1}^{n} \alpha_{i,j} \mathbf{y}_j$

where, d_j is the degree of the j-th node in i-th node's neighbourhood, and, α represent the soft-maxed attention coefficients. From this intermediate formulation, we can determine that the output's mean and variance of the aggregation stage for GIN layers should grow linearly with the node's degree, $\mathcal{O}(n)$. Similarly, the growth for GCN layers is $\mathcal{O}(\sqrt{n})$ since each node's message is weighted inversely proportionally to the square root of its degree. On the other hand, the normalized attention mechanism in GAT networks makes the accumulation during aggregation to be $\mathcal{O}(1)$.

We empirically validate these predictions on networks trained on the Cora dataset, a single-graph dataset with a maximum node degree $n = 168$. In Figure 5.3 (top row) we show how the distribution of the output of the aggregation stage $\mathbf{o}_i^{(n)}$ varies as a function of the node degree. We see that the aggregation values do follow the trends predicted: for GIN, the aggregation grows rapidly with the node degree; this growth is less severe with GCN since each message is normalized with the product of degrees along each edge; finally, the normalized attention in GAT keeps these distributions almost constant. From this empirical evidence, it would be expected that GIN layers are most affected by quantization since the number of bits needed to represent the wide spectrum of values observed in the output of the aggregation is more than double compared to GCN or GAT. At the same bitwith, e.g. 8 bits, the numerical resolution for GIN would be considerably lower than for the other architectures. The impact of quantization would be specially damaging on graphs following a power-law distribution or other graph distributions that, even though closer to a normal or uniform degree distribution, higher degree nodes carry important semantic meaning. This is the case for example in molecular graphs (e.g. ZINC), where nodes represent atoms and edges features carry information of the particular bond type. If quantization

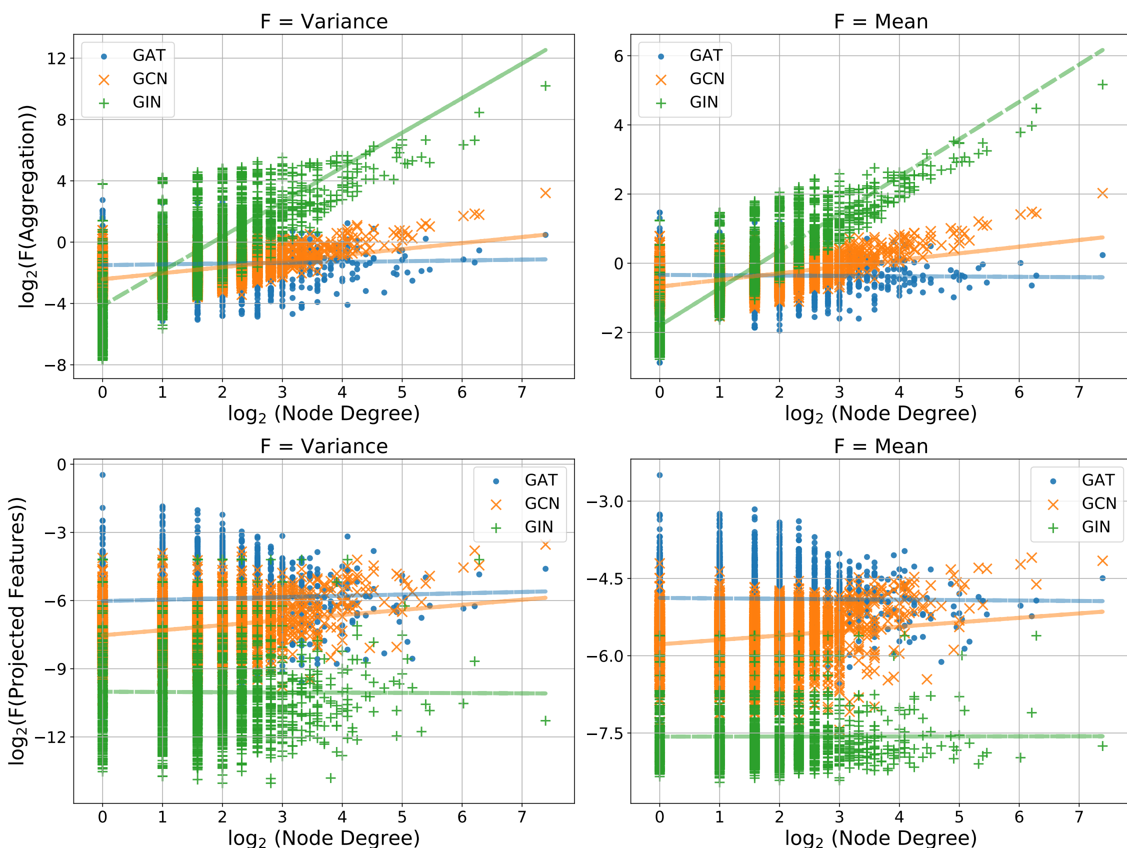


Figure 5.3: Analysis of values collected immediately after aggregation (top two plots) and after generating node embeddings (bottom row) at the final layer of FP32 GNNs trained on Cora's validation set. The data shown has been averaged over 100 runs. For each architecture, we show the mean and variance of the aggregated messages, $\mathbf{O} \in \mathbb{R}^{|V| \times F'}$, at each node and represent it as a point in the plot. We follow the same approach to represent the node embeddings, $\mathbf{Y} \in \mathbb{R}^{|V| \times F'}$. **Top row:** As degree grows, so does the mean and variance of channel values after aggregation except in GAT. This suggest that quantization of the aggregation stage would have a more damaging effect on GIN, less on GCN and, negligible in GAT models. **Bottom row:** The features' distributions remain within an almost constant range of values independently of the node's degree. The dashed line is the result of performing a Ridge regression on the results for each architecture.

introduces sufficient error in the representation of only a single node, the properties of the molecule representing the graph might change entirely. In Figure 5.3 (bottom row plots) we empirically show that growth observed during aggregation cannot be attributed to the node embeddings themselves, since these remain within constant ranges across the entire graph. This further evidences that the major contributor to quantization degradation is the aggregation mechanism and, the normalization stage applied during message generation.

Quantization errors introduced during the aggregation stage will have an impact on the quality of the learned weights. We can derive the impact that this stage has in the training process by taking the derivative of the loss with respect to the layer weights:

GIN GCN

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} = \sum_{i=1}^{|V|} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{h}_i^{l+1}} \circ f'(\mathbf{W}^l \mathbf{o}_i^l) \right) \mathbf{o}_i^{l^\top} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \sum_{i=1}^{|V|} \sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{d_i d_j}} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{h}_i^{l+1}} \circ f'(\mathbf{o}_i^l) \right) \mathbf{h}_j^{l^\top}$$

The larger the error in $\mathbf{o}_i$ —caused by aggregation error— the greater the error in the weight gradients for GIN, which results in poorly performing models being obtained. In other words, high degree nodes would not only accumulate more error due to quantization but also spread this error to all their neighbourhing nodes during backpropgataion. The same argument applies to GCN, with the $\mathbf{h}_j^{l^\top}$ and $\mathbf{o}_i^l$ terms introducing aggregation error into the weight updates.

5.4 A topology-aware Quantization Scheme for GNNs

From the analysis and empirical observations in Section 5.3, an attempt could be made to design a quantization framework that assigns different bitwidths to each node (or cluster of nodes) based on their degree. In this way, each row in $\mathbf{O} \in \mathbb{R}^{|V| \times F'}$ would be quantized independently. Such framework would resemble other quantization frameworks present in the literature that follow an importance-based optimization stage to determine which elements in the model (e.g. layers, channels) need to be kept at higher precision [237, 101]. Although these frameworks achieve high compression rations, in some cases with minimal degradation in model performance, they are difficult to accelerate in general purpose hardware. We instead seek to provide a general re-formulation of quantization-aware training for graphs that allows the acceleration of quantized GNNs on off-the-shelf hardware.

To address these sources of error we propose `Degree-Quant` (DQ), a method for QAT with GNNs where all elements in the message-passing pipeline are quantized to the same bitwidth at inference time. In this section we consider both *inaccurate weight updates* and *unrepresentative quantization ranges* and, present solutions to each of these challenges that manifest in a unique way in graphs.

5.4.1 Topology-aware Masking of Graph Neural Networks

`Degree-Quant` aims to encourage more accurate weight updates by stochastically protecting nodes in the graph from quantization. At each layer and training iteration, a protective node mask is generated; all masked nodes have the phases of the message generation, aggregation and update performed at full precision (FP32). This includes messages sent

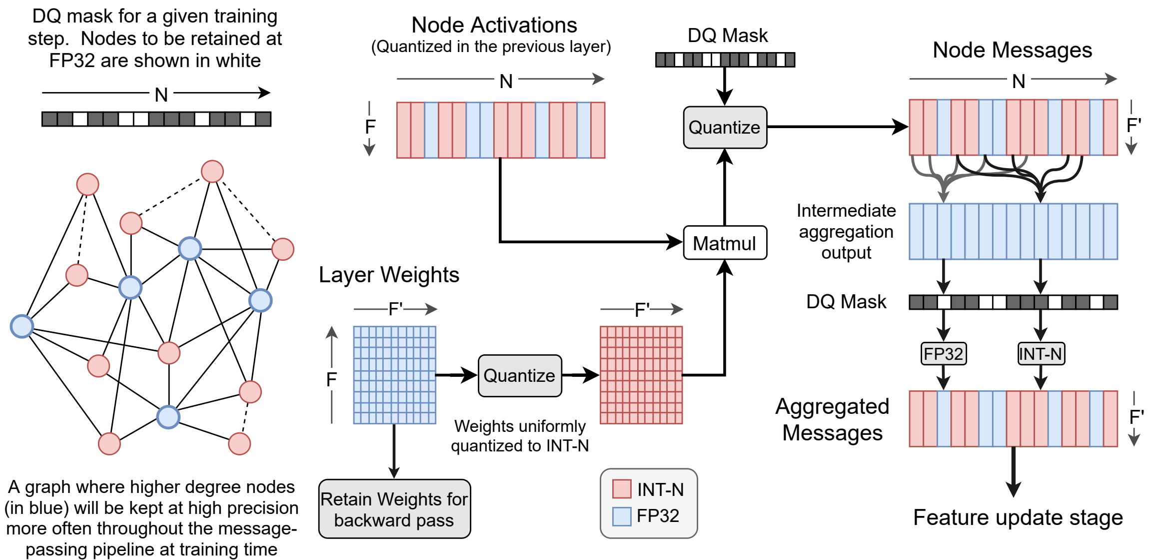


Figure 5.4: High-level view of the stochastic element of `Degree-Quant`. Protected (high degree) nodes, in blue, operate at full precision more often, while unprotected nodes (red) operate at reduced precision. After each of the three stages (update stage not show in the diagram) in the message-passing pipeline, the DQ mask is applied to ensure that these can be accelerated using integer arithmetic. High degree nodes contribute most to poor gradient estimates, hence they are stochastically protected from quantization more often. After training, all stages are quantized to the same bitwidth, including high degree nodes. Figure A.7 shows an equivalent diagram for QAT.

by protected nodes to other nodes, as shown in Figure 5.4. It is also important to note that the weights used at all nodes are the same quantized weights; this is motivated by the fact that our method is used to encourage more accurate gradients to flow back to the weights through high degree nodes. At test time the protective masking is disabled and all nodes operate at the same reduced bitwidth. More formally, the forward pass of training GNNs using DQ is show in Algorithm 2.

To generate the mask, we pre-process each graph before training and create a vector of probabilities $\mathbf{p}$ with length equal to the number of nodes. At training time, a mask $\mathbf{m}$ is generated by sampling using the Bernoulli distribution: $\mathbf{m} \sim \text{Bernoulli}(\mathbf{p})$. In our scheme p_i is higher if the degree of the i-th node is large, as we empirically found that high degree nodes contribute most towards error in weight updates. Our scheme relies on two hyperparameters, p_{min} and p_{max}; nodes with the maximum degree are assigned p_{max} as their masking probability, with all other nodes assigned a probability calculated by interpolating between p_{min} and p_{max} based on their degree ranking in the graph.

Algorithm 2: Training with `Degree-Quant` (DQ). Functions accepting a protective mask $\mathbf{m}$ perform only the masked computations at full precision: intermediate tensors are *not* quantized. At test time the protective masking is disabled.

```
1:  procedure TRAINFORWARDPASS(𝒢, p)
2:      ▷ Calculate mask and quantized weights, Θ′, which all operations share
3:      m ← BERNOULLI(p)
4:      ▷ Quantize using off-the-shelf QAT
5:      Θ′ ← QUANTIZE(Θ)
6:      ▷ Messages with masked sources are at full precision
7:      ℳ ←MESSAGECALCULATE(𝒢, Θ′, m)
8:      ▷ While accumulation happens at FP32, the result gets quantized according to m
9:      X ← QUANTIZE(AGGREGATE(ℳ, Θ′, m), m)
10:     return UPDATE(X, Θ′, m)
11: end procedure
```

5.4.2 Percentile Tracking of Quantization Ranges

Figure 5.3 demonstrates large fluctuations in the variance of the aggregation output as degree increases. Since these can disproportionately affect the ranges found by using min-max or momentum-based quantization, we propose using *percentiles*. While percentiles have been used for post-training quantization [372], this work is the first (to the best of our knowledge) to propose making it a core part of QAT; we find it to be a key contributor to achieving consistent results with graphs. Using percentiles involves ordering the values in the tensor and clipping a fraction of the values at both ends of the distribution. The fraction to clip is a hyperparameter of each layer. In our formulation, we are more aggressive than existing literature on the quantity we discard: we clip the top and bottom 0.1%, rather than 0.01%, as we observe the fluctuations to be a larger issue with GNNs than with CNNs or DNNs. The resulting clipped quantization ranges are more representative of the vast majority of values in this scheme, resulting in less *rounding error*.

We emphasize that a core contribution of DQ is that it is *architecture-agnostic*. Our method enables a wide variety of architectures to use low precision arithmetic at inference time. `Degree-Quant` method is also orthogonal—and complementary—to other techniques for decreasing GNN computation requirements, such as sampling based methods which are used to reduce memory consumption [405, 138], or weight pruning [39] approaches to achieve further model compression and acceleration.

Model Arch.	# Layers					# Hidden Units					Residual					Output MLP				
	Cit	M	C	Z	R	Cit	M	C	Z	R	Cit	M	C	Z	R	Cit	M	C	Z	R
GCN	2	4	4	4	-	16	146	146	145	-	×	✓	✓	✓	-	×	✓	✓	✓	-
GAT	2	4	4	4	-	8	19	19	18	-	×	✓	✓	✓	-	×	✓	✓	✓	-
GIN	2	4	4	4	5	16	110	110	110	64	×	✓	✓	✓	×	×	✓	✓	✓	✓

Table 5.1: Configuration of architectures evaluated for citation networks (Cit), MNIST (M), CIFAR-10 (C), ZINC (Z) and REDDIT-BINARY (R). We relied on Adam optimizer for all experiments. For all batched experiments, we used 128 batch-sizes. All GAT models used 8 attention heads. All GIN architectures used 2-layer MLPs, except those for citation networks which used a single linear layer.

5.5 Experimental Setup

As baselines we use the architectures and results reported by Fey & Lenssen [105] for citation networks, those in Dwivedi et al. [95] for MNIST, CIFAR-10 and ZINC and, Xu et al. [381] for REDDIT-BINARY. We re-implemented the architectures used in these publications and replicated the results reported at FP32. For models using GIN layers, we learn parameter ϵ. These instantiation of GIN layers are often referred as to GIN-ϵ, but to keep notation concise we refer to them as simply GIN. The high-level description of these architectures is shown in Table 5.1. The number of parameters for each architecture-dataset pair in this work are shown in Table 5.2.

Our infrastructure was implemented using PyTorch Geometric (PyG) [105]. We generate candidate hyperparameters using random search, and prune trials using the asynchronous hyperband algorithm [219]. Hyperparameters searched over were learning rate, weight decay, dropout [307] and drop-edge [289] probabilities. The search ranges were initialized centered at the values used in the reference implementations of the baselines. `Degree-Quant` requires searching for two additional hyperparameters, $p_{\min}$ and $p_{\max}$, these were tuned in a grid-search fashion. We report our results using the hyperparameters which achieved the best validation loss over 100 runs on the Cora and Citeseer datasets, 10 runs for MNIST, CIFAR-10 and ZINC, and 10-fold cross-validation for REDDIT-BINARY.

We generally used fewer hyperparameter runs for DQ runs than for baselines, even ignoring the searches over the various STE configs. As our method is more stable, finding a reasonable set of parameters was easier than before. As is usual with quantization experiments, we found it useful to decrease the learning rate relative to the FP32 baseline.

For QAT experiments, all elements of each network are quantized: inputs to each layer, the weights, the messages sent between nodes, the inputs to aggregation stage and its outputs and, the outputs of the update stage (which are the outputs of the GNN layer before activation). In this way, all intermediate tensors in GNNs are quantized with the exception

Model Arch.	Node Classification Cora	Citeseer	Graph Classification MNIST	CIFAR-10	REDDIT-BIN	Graph Regression ZINC
GCN	23063	59366	103889	104181	-	105454
GAT	92373	237586	113706	114010	-	105044
GIN	23216	59536	104554	104774	42503	102088

Table 5.2: Number of parameters for each of the evaluated architectures

Dataset	Graphs	Nodes	Edges	Features	Labels	Task
Cora	1	2,708	5,278	1,433	7	NC
Citeseer	1	3,327	4,552	3,703	6	NC
Pubmed	1	19,717	44,324	500	3	NC
MNIST	70K	40-75	564.53 (avg)	3	10	GC
CIFAR10	60K	85-150	941.07 (avg)	5	10	GC
ZINC	12K	9-37	49.83 (avg)	28	1	GR
REDDIT-BINARY	2K	429.63 (avg)	497.75 (avg)	1	2	GC
Reddit	1	232,965	114,848,857	602	41	NC
COLLAB	1	235,868	1,285,465	128	—	LP

Table 5.3: Statistics for a few popular datasets found in the literature. Acronyms for tasks: NC, stands for Node Classification; GC, stands for Graph Classification; GR, stands for Graph Regression; LP, stands for Link Prediction.

of the attention mechanism in GAT; we do not quantize after the softmax calculation, due to the numerical precision required at this stage. With the exception of Cora and Citeseer, the models evaluated in this work make use of Batch Normalization [174]. For deployments of quantized models, Batch Normalization layers are often *folded* with the weights [198]. This is to ensure the input to the next layer is within the expected $[q_{\min}, q_{\max}]$ ranges. In this work, for both QAT baselines and QAT+DQ, we left BN layers unfolded but ensure the inputs and outputs were quantized to the appropriate number of bits (i.e. INT8 or INT4) before getting multiplied with the layer weights. We leave as future work proposing a BN folding mechanism applicable for GNNs and studying its impact for deployments of quantized GNNs.

The GIN models evaluated on REDDIT-BINARY used QAT for all layers with the exception of the input layer of the MLP in the first GIN layer. This compromise was needed to overcome the severe degradation introduced by quantization when operating on nodes with a single scalar as feature, as it is the case with this dataset.

In addition to QAT, we also consider the use of its stochastic variant as implemented in Fan et al. [309]. At each training step a binary mask sampled from a Bernoulli distribution is used to specify which elements of the weight tensor will be quantized and which will be left at full precision. We experimented with block sizes larger than one (i.e. a single

scalar) but often resulted in a sever drop in performance, further highlighting that borrowing quantization techniques from CNNs is not straight forward in most cases. All the reported results therefore use block size of one. We refer to this approach as *noisy* QAT or nQAT.

5.5.1 Datasets and Tasks

We show in Table 5.3 the statistics for each dataset used to validate `Degree-Quant` and others that have been referenced prior in this chapter. For Cora and Citeseer datasets, nodes correspond to documents and edges to citations between these. Node features are a bag-of-words representation of the document. The task is to classify each node in the graph (i.e. each document) correctly. The image classification MNIST and CIFAR-10 datasets are transformed using SLIC [7] into graphs where each node represents a cluster of perceptually similar pixels or superpixels. The task is to classify each image using their superpixels graph representation. The ZINC dataset contains graphs representing molecules, where each node represents an atom. The task is to regress a molecular property (constrained solubility [183]) given the graph representation of the molecule. This dataset also includes edge features that specify the type of bond between atoms, however, these are not taken into account during learning in this work. Nodes in graphs of the REDDIT-BINARY dataset represent users of a Reddit thread with edges drawn between a pair of nodes if these interacted. This dataset contains graphs of two types of communities: question-answer threads and discussion threads. The task is to determine if a given graph is from a question-answer thread or a discussion thread. Intuitively, these two types of social interactions follow a different pattern, which should be reflected on the graph topology.

We use standard splits for MNIST, CIFAR-10 and ZINC. For citation datasets, we use the splits used by [192]. For REDDIT-BINARY we use 10-fold cross validation. We refer the reader to [105] for further information on the Cora, Citeseer and REDDIT-BINARY datasets and, to [95] for MNIST, CIFAR-10 and ZINC datasets.

5.6 Experimental Evaluation

In this section we first analyze how the choice of quantization implementation affects performance of GNNs. We subsequently evaluate `Degree-Quant` against the strong baselines of: FP32, INT8-QAT and, INT8-QAT with stochastic masking of weights that we name INT8-nQAT. To make explicit that we are quantizing both weights and activations, we use the notation W8A8. We repeat these experiments at INT4. Our study evaluates performance on six datasets and includes both node-level and graph-level tasks. Across

	Model Arch.	*vanilla* STE min/max W8A8	*vanilla* STE min/max W4A4	*vanilla* STE momentum W8A8	*vanilla* STE momentum W4A4	STE with Gradient Clipping min/max W8A8	STE with Gradient Clipping min/max W4A4	STE with Gradient Clipping momentum W8A8	STE with Gradient Clipping momentum W4A4
CORA	GCN	$\mathbf{81.0 \pm 0.7}$	65.3 ± 4.9	42.3 ± 11.1	49.4 ± 8.8	80.8 ± 0.8	62.3 ± 5.2	66.9 ± 18.2	$\mathbf{77.2 \pm 2.5}$
	GAT	76.0 ± 2.2	16.8 ± 8.5	81.7 ± 1.3	$\mathbf{51.7 \pm 5.8}$	76.4 ± 2.6	15.4 ± 8.1	$\mathbf{81.9 \pm 0.7}$	47.4 ± 5.0
	GIN	69.9 ± 1.9	25.9 ± 2.6	49.2 ± 10.2	$\mathbf{42.8 \pm 4.0}$	69.2 ± 2.3	29.5 ± 3.5	$\mathbf{75.1 \pm 1.1}$	40.5 ± 5.0
MNIST	GCN	$\mathbf{90.4 \pm 0.2}$	51.3 ± 7.5	90.1 ± 0.5	$\mathbf{70.6 \pm 2.4}$	90.4 ± 0.3	54.8 ± 1.5	90.2 ± 0.4	10.3 ± 0.0
	GAT	$\mathbf{95.8 \pm 0.1}$	20.1 ± 3.3	95.7 ± 0.3	67.4 ± 3.2	95.7 ± 0.1	30.2 ± 7.4	95.7 ± 0.3	$\mathbf{76.3 \pm 1.2}$
	GIN	96.5 ± 0.3	62.4 ± 21.8	$\mathbf{96.7 \pm 0.2}$	$\mathbf{91.0 \pm 0.6}$	96.4 ± 0.4	19.5 ± 2.1	75.3 ± 18.1	10.8 ± 0.9
ZINC	GCN	0.49 ± 0.0	0.75 ± 0.0	0.51 ± 0.0	0.71 ± 0.1	0.46 ± 0.0	0.77 ± 0.0	$\mathbf{0.48 \pm 0.0}$	$\mathbf{0.69 \pm 0.0}$
	GAT	0.47 ± 0.0	0.74 ± 0.0	0.57 ± 0.0	$\mathbf{0.69 \pm 0.1}$	0.47 ± 0.0	0.76 ± 0.0	$\mathbf{0.46 \pm 0.0}$	0.72 ± 0.0
	GIN	0.39 ± 0.0	1.21 ± 0.3	$\mathbf{0.39 \pm 0.0}$	$\mathbf{0.57 \pm 0.0}$	0.39 ± 0.0	1.67 ± 0.1	0.39 ± 0.0	0.97 ± 0.2

Table 5.4: Impact on performance of four typical quantization implementations for INT8 and INT4. The configuration that resulted in best performing models for each dataset-model pair is bolded. Hyperparameters for each experiment were fine-tuned independently. As expected, adding clipping does not change performance with min/max but does with momentum.

all datasets INT8 models trained with `Degree-Quant` manage to recover most of the accuracy lost as a result of quantization. In some instances, DQ-INT8 outperform the extensively tuned FP32 baselines. For INT4, DQ outperforms all QAT baselines and results in double digits improvements over QAT-INT4 in some settings.

5.6.1 Performance with off-the-shelf Quantization Approaches

The STE is a workaround for when the forward pass contains non-differentiable operations (e.g. rounding in QAT) that has been widely adopted in practice. While the choice of STE implementation generally results in marginal differences for CNNs—even for binary networks [11]—it is unclear whether only marginal differences will also be observed for GNNs. Motivated by this, we study the impact of four off-the-shelf quantization procedures on the three architectures evaluated for each type of dataset; the implementation details of each one is described in Table 5.1. We perform this experiment to ensure that we have the strongest possible QAT baselines. Results are shown in Table 5.4. We found the choice quantization implementation to be highly dependent on the model architecture and type of problem to be solved: we see a much larger variance than is observed with CNNs; this is an important discovery for future work building on our study.

We observe a general trend in all INT4 experiments benefiting from momentum as it helps smoothing out the quantization statistics for the inherently noisy training stage at low bitwidths. This trend applies as well for the majority of INT8 experiments, while exhibiting little impact on MNIST. For INT8 Cora-GCN, large gradient norm values in the early stages of training (see blue line in Figure 5.5) mean that these models do not benefit from momentum as quantization ranges fail to keep up with the rate of changes in tensor values;

Quant. Scheme	Model Arch.	Node Classification (Accuracy %)		Graph Classification (Accuracy %)		Graph Regression (Loss)
		Cora ↑	Citeseer ↑	MNIST ↑	CIFAR-10 ↑	ZINC ↓
Ref. (FP32)	GCN	81.4 ± 0.7	71.1 ± 0.7	90.0 ± 0.2	54.5 ± 0.1	0.469 ± 0.002
	GAT	83.1 ± 0.4	72.5 ± 0.7	95.6 ± 0.1	65.4 ± 0.4	0.463 ± 0.002
	GIN	77.6 ± 1.1	66.1 ± 0.9	93.9 ± 0.6	53.3 ± 3.7	0.414 ± 0.009
Ours (FP32)	GCN	81.2 ± 0.6	71.4 ± 0.9	90.9 ± 0.4	58.4 ± 0.5	0.450 ± 0.008
	GAT	83.2 ± 0.3	72.4 ± 0.8	95.8 ± 0.4	65.1 ± 0.8	0.455 ± 0.006
	GIN	77.9 ± 1.1	65.8 ± 1.5	96.4 ± 0.4	57.4 ± 0.7	0.334 ± 0.024
QAT (W8A8)	GCN	81.0 ± 0.7	71.3 ± 1.0	90.9 ± 0.2	56.4 ± 0.5	0.481 ± 0.029
	GAT	81.9 ± 0.7	71.2 ± 1.0	95.8 ± 0.3	66.3 ± 0.4	0.460 ± 0.005
	GIN	75.6 ± 1.2	63.0 ± 2.6	96.7 ± 0.2	52.4 ± 1.2	0.386 ± 0.025
nQAT (W8A8)	GCN	81.0 ± 0.8	70.7 ± 0.8	91.1 ± 0.1	56.2 ± 0.5	0.472 ± 0.015
	GAT	82.5 ± 0.5	71.2 ± 0.7	96.0 ± 0.1	66.7 ± 0.2	0.459 ± 0.007
	GIN	77.4 ± 1.3	65.1 ± 1.4	96.4 ± 0.3	52.7 ± 1.4	0.405 ± 0.016
DQ (W8A8)	GCN	81.7 ± 0.7 (**+0.7**)	71.0 ± 0.9 (**-0.3**)	90.9 ± 0.2 (**-0.2**)	56.3 ± 0.1 (**-0.1**)	0.434 ± 0.009 (**+9.8**)
	GAT	82.7 ± 0.7 (**+0.2**)	71.6 ± 1.0 (**+0.4**)	95.8 ± 0.4 (**-0.2**)	67.7 ± 0.5 (**+1.0**)	0.456 ± 0.005 (**+0.9**)
	GIN	78.7 ± 1.4 (**+1.3**)	67.5 ± 1.4 (**+2.4**)	96.6 ± 0.1 (**-0.1**)	55.5 ± 0.6 (**+2.8**)	0.357 ± 0.014 (**+7.5**)
QAT (W4A4)	GCN	77.2 ± 2.5	64.1 ± 4.1	70.6 ± 2.4	38.1 ± 1.6	0.692 ± 0.013
	GAT	55.6 ± 5.4	65.3 ± 1.9	76.3 ± 1.2	41.0 ± 1.1	0.655 ± 0.032
	GIN	42.5 ± 4.5	18.6 ± 2.9	91.0 ± 0.6	45.6 ± 3.6	0.572 ± 0.02
nQAT (W4A4)	GCN	78.1 ± 1.5	65.8 ± 2.6	70.9 ± 1.5	40.1 ± 0.7	0.669 ± 0.128
	GAT	54.9 ± 5.6	65.5 ± 1.7	78.4 ± 1.5	41.0 ± 0.6	0.637 ± 0.012
	GIN	45.0 ± 5.0	34.6 ± 3.8	91.3 ± 0.5	48.7 ± 1.7	0.561 ± 0.068
DQ (W4A4)	GCN	78.3 ± 1.7 (**+0.2**)	66.9 ± 2.4 (**+1.1**)	84.4 ± 1.3 (**+13.5**)	51.1 ± 0.7 (**+11.0**)	0.536 ± 0.011 (**+26.2**)
	GAT	71.2 ± 2.9 (**+16.3**)	67.6 ± 1.5 (**+2.1**)	93.1 ± 0.3 (**+14.7**)	56.5 ± 0.6 (**+15.5**)	0.520 ± 0.021 (**+20.6**)
	GIN	69.9 ± 3.4 (**+24.9**)	60.8 ± 2.1 (**+26.2**)	95.5 ± 0.4 (**+4.2**)	50.7 ± 1.6 (**+2.0**)	0.431 ± 0.012 (**+23.2**)

Table 5.5: This table is divided into three sets of rows with FP32 baselines at the top. We provide two baselines for INT8 and INT4: standard QAT and stochastic QAT (nQAT). Both are informed by the analysis in 5.6.1, with nQAT achieving better performance in some cases. Models trained with `Degree-Quant` (DQ) are always comparable to baselines, and usually substantially better, especially for INT4.

higher momentum can help but also leads to instability. In contrast, GAT has stable initial training dynamics, and hence obtains better results with momentum. For the molecules dataset ZINC, we consistently obtained lower regression loss when using momentum. We note that GIN models often suffer from higher performance degradation (as was first noted in Figure 5.3), specially at W4A4. This is not the case however for image datasets using superpixels. As argued earlier, datasets with Gaussian-like node degree distributions are more tolerant of the imprecision introduced by quantization, compared to datasets with tailed distributions.

5.6.2 Obtaining Quantization Baselines

Our FP32 results, which we obtain after extensive hyperparameter tuning, and those from the baselines are shown at the top of Table 5.5. We observed large gains on MNIST, CIFAR10 and, ZINC compared to the baselines.

Quantization	REDDIT-BIN
Ref. (FP32)	92.2 ± 2.3
Ours (FP32)	92.0 ± 1.5
QAT-W8A8	76.1 ± 7.5
nQAT-W8A8	77.5 ± 3.4
DQ-W8A8	91.8 ± 2.3 (**+14.3**)
QAT-W4A4	54.4 ± 6.6
nQAT-W4A4	58.0 ± 6.3
DQ-W4A4	81.3 ± 4.4 (**+23.0**)

Table 5.6: Results for DQ-INT8 GIN models perform nearly as well as at FP32. For INT4, DQ offers a significant increase in accuracy of over 23% compared to the stochastic QAT baseline.

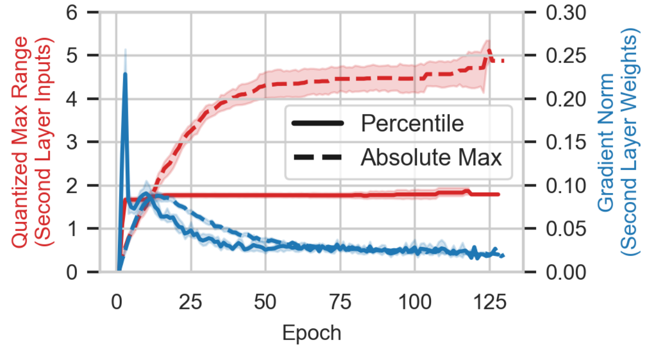


Figure 5.5: Maximum observable values with absolute `min`/`max` and percentile ranges, applied to INT8 GCN training on Cora. We observe that the percentile max is half that of the absolute method, *doubling* resolution for the majority of values.

For our QAT-INT8 and QAT-INT4 baselines, we use the quantization configurations informed by our analysis in Section 5.6.1. For Citeseer we use the best resulting setup analyzed for Cora, and for CIFAR-10 that from MNIST. Then, the hyperparameters for each experiment were fine tuned individually, including noise rate $n \in [0.5, 0.95]$ for nQAT experiments. QAT-INT8 and QAT-INT4 results in Table 5.5, with the exception of MNIST (an easy to classify dataset), corroborate our hypothesis that GIN layers are less resilient to quantization. This was first observed in Figure 5.3. In the case of ZINC, while all models results in noticeable degradation, GIN sees a more severe 16% increase of regression loss compared to our FP32 baseline. For QAT W4A4 an accuracy drop of over 35% and 47% is observed for Cora and Citeseer respectively. The stochasticity induced by nQAT helped in recovering some of the accuracy lost as a result of quantization for citation networks (both INT8 and INT4) but had little impact on other datasets and harmed performance in some cases. Similarly, in the case of GIN for REDDIT-BINARY, both QAT and nQAT result in severe accuracy degradation, as shown inTable 5.6.

5.6.3 Performance with `Degree-Quant`

`Degree-Quant` provides superior quantization for all GNN datasets and architectures. Our results with DQ are highlighted in gray in Table 5.5 and Table 5.6. Citation networks trained with DQ for W8A8 manage to recover most of the accuracy lost as a result of QAT and outperform most of nQAT baselines. In some instances DQ-W8A8 models outperform the reference FP32 baselines. At 4-bits, DQ results in even larger gains compared to W4A4 baselines. We see DQ being more effective for GIN layers, outperforming INT4 baselines for Cora (+24.9%), Citeseer (+26.2%) and REDDIT-BINARY (+23.0%) by large margins.

Device	Arch.	Zinc (Batch=10K)			CIFAR-10 (Batch=1K)			Reddit		
		FP32	W8A8	Speedup	FP32	W8A8	Speedup	FP32	W8A8	Speedup
CPU	GCN	181ms	42ms	4.3×	182ms	88ms	2.1×	13.1s	3.1s	4.2×
	GAT	190ms	50ms	3.8×	500ms	496ms	1.0×	13.1s	2.8s	4.7×
	GIN	182ms	43ms	4.2×	144ms	44ms	3.3×	13.1s	3.1s	4.2×
GPU	GCN	39ms	31ms	1.3×	2.1ms	1.6ms	1.3×	191ms	176ms	1.1×
	GAT	17ms	15ms	1.1×	30.0ms	27.1ms	1.1×	OOM	OOM	-
	GIN	39ms	31ms	1.3×	20.9ms	16.2ms	1.2×	191ms	176ms	1.1×

Table 5.7: INT8 latency results run on a 22 core 2.1GHz Intel Xeon Gold 6152 and, on a GTX 1080Ti GPU. Quantization provides large speedups on a variety of graphs for CPU and non-negligible speedups with unoptimized INT8 GPU kernels.

Models trained with DQ at W4A4 for graph classification and graph regression also exhibit large performance gains (of over 10%) in most cases. For ZINC, all models achieve over 20% lower regression loss. Among the top performing models using DQ, ratios of $p_{\min}$ and $p_{\max}$ in $[0.0, 0.2]$ were the most common. Figure A.5 in the Appendix shows the validation curves for GIN models trained using different $p_{\min}$ and $p_{\max}$ on the REDDIT-BIN dataset.

Figure 5.5 shows the benefit of using percentiles during training. We see that when using absolute `min`/`max` tracking of quantization statistics, the upper range grows to over double the range required for 99.9% of values, effectively halving the resolution of the quantized values. With regards to gradient clipping, we found it had no clear benefits when combined with percentiles: all results used the STE, with the exception of REDDIT-BINARY. Overall, `Degree-Quant` is more stable and required over an order of magnitude less tuning relative to the baselines.

5.6.4 Latency and Memory Implications

In addition to offering significantly lower memory usage ($4\times$ with INT8), quantization can reduce latency—especially on CPUs. We found that with INT8 arithmetic we could accelerate inference by up to $4.7\times$. We note that the latency benefit depends on the graph topology and feature dimension, therefore we ran benchmarks on a variety of graph datasets, including Reddit[2], Zinc, and CIFAR-10; results are shown in Table 5.7. For a GCN layer with in- and out-dimension of 128, we obtain speed-ups of: $4.3\times$ on Reddit, $2.5\times$ on Zinc and, $2.1\times$ on CIFAR-10. It is also worth emphasizing that quantized networks are necessary to efficiently use accelerators deployed in smartphones and smaller devices as they primarily accelerate integer arithmetic, and that CPUs remain a common choice for model serving on servers. The decrease in latency on CPUs is due to improved cache performance

[2]The largest graph commonly benchmarked on in the GNN literature

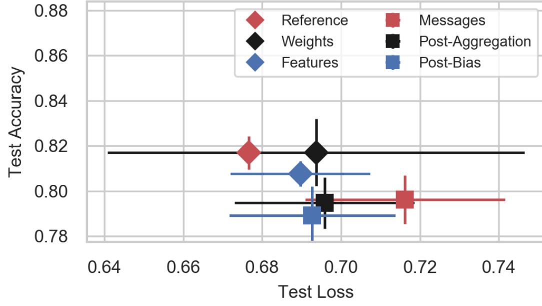


Figure 5.6: Degradation of INT8 GCN on Cora as individual elements are converted to INT4 *without* `Degree-Quant`. No single stage has a large impact in final accuracy when at INT4.

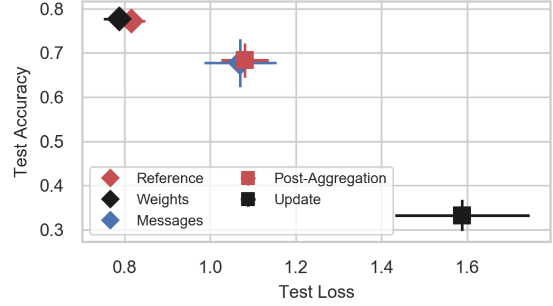


Figure 5.7: Degradation of INT8 GIN on Cora as individual elements are converted to INT4 *without* `Degree-Quant`. Quantizing the `update` stage results in severe degradation.

Quantization Scheme	Model Arch.	Node Classification Cora ↑	Citeseer ↑	Graph Regression ZINC ↓
QAT-INT8 + DQ Masking	GCN	81.1 ± 0.6 (**+0.1**)	71.0 ± 0.7 (**-0.3**)	0.468 ± 0.014 (**+2.7**)
	GAT	82.1 ± 0.1 (**+0.2**)	71.4 ± 0.8 (**+0.1**)	0.462 ± 0.005 (**-0.4**)
	GIN	78.9 ± 1.2 (**+3.3**)	67.1 ± 1.7 (**+4.1**)	0.347 ± 0.028 (**+10.1**)
QAT-INT4 + DQ Masking	GCN	78.5 ± 1.4 (**+1.3**)	62.8 ± 8.5 (**-1.3**)	0.599 ± 0.015 (**+13.4**)
	GAT	64.4 ± 9.3 (**+8.8**)	68.9 ± 1.2 (**+3.6**)	0.529 ± 0.008 (**+19.2**)
	GIN	71.2 ± 2.9 (**+28.7**)	56.7 ± 3.8 (**+38.1**)	0.427 ± 0.010 (**+25.3**)

Table 5.8: Ablation study against the two elements of `Degree-Quant` (DQ). The first two rows of results are obtained with only the stochastic element of DQ enabled for INT8 and INT4. Percentile-based quantization ranges are disabled in these experiments. The relative improvement is compared against vanilla QAT implemenation. DQ masking alone is often sufficient to achieve excellent results, specially in the case of GIN, but the addition of percentile-based range tracking can be beneficial to increase stability.

for the sparse operations; GPUs, however, see less benefit due to their massively-parallel nature which relies on mechanisms other than caching to hide slow random memory accesses, which are unavoidable in this application.

5.7 Discussion

In this section we present two ablation studies to further asses the impact of each of the elements in `Degree-Quant` and, perform an initial analysis on the sensitivity of each element in the message-passing pipeline to INT4 quantization. We then propose ways to improve the framework presented in this chapter as well as some thoughts on future work.

Source of degradation at INT4. With `Degree-Quant`, GNNs trained with INT8 quantization perform as well as the FP32 baselines in most cases. However, at INT4, the degradation is substantially higher and, in some cases, only partially minimized in DQ. In order to understand the precipitous drop in accuracy in the INT4 baselines, in Figure 5.6 and Figure 5.7 we assess how INT8 models without DQ degrade as single elements are converted to INT4 while the rest is kept at INT8. We observe that GCN is fairly tolerant to INT4 quantization. GIN, however, requires accurate representations after the update stage, and heavily suffers from further quantization. A similar patter in observed for GAT layers, as shown in Figure A.4 in the Appendix. Interestingly, common to all architectures is the fact that weights seems to tolerate quantization without inducing performance drops, leaving room for further optimizations. An option to reduce the impact of INT4 in the `update` stage in GAT and GIN layers would be to keep them at different numerical resolutions. The idea of performing different stages of inference at different precisionz has been proposed, although it is uncommon [347]. Instead we argue that making uses of more sophisticated quantization implementations, such as non-uniform quantization, while maintaining the same bitwidth, would be preferable.

Impact of the stochastic element of `Degree-Quant`. One of the core elements of `Degree-Quant` is the stochastic masking of nodes based on their degree. One question that has not beed directly address is: *what is the impact of the protective masking alone without considering the use of percentiles?* In Table 5.8 we show how the stochastic masking often achieves most of the performance gain over the QAT baseline. The benefit of the percentile-based quantization ranges is *stability*, although it can yield some performance gains. The full DQ method provides consistently good results on all architectures and datasets considered, without requiring an extensive analysis as discussed in Section 5.6.1 and shown in Table 5.4.

Quantization-immune aggregation stages. In the case of standard convolutional layers found in CNNs, despite minor implementation level choices (e.g. striding, dilation, etc) all convolutional layers *stay true* to the convolutional or cross-correlation operation. As a consequence, the various existing quantization approaches can be easily ported across architectures. In this chapter we have identified the sources of degradation for three common GNN layer architectures. However, as first introduced in Section 2.5, for GNNs it is up to each specific layer design how the aggregation stage is implemented. In many cases this is their key differential and, the reason why some architectures excel at certain applications

(e.g. GIN for molecular regression on the ZINC dataset) but not for others (e.g. GIN for citation networks). This key difference between CNN and GNNs raises the question: *is there a type of GNN layer that is better suited for quantization?* or more concretely, *is there an aggregation stage that is inherently more robust to quantization?* From the analysis in Section 5.3, we already have a intuition of a key component needed for an aggregation strategy to perform well: normalization of messages helps to keep variance of observed values under control. Another approach could involve adding a more sophisticated node-level stage (currently most layers perform a single `matmul`), with the aim of keeping the graph-level aggregation stage simple and, therefore facilitate the adoption of quantization methods that map well to the available hardware resources.

5.8 Summary

This chapter proposed the first general formulation for quantization-aware training on graphs. We identified the sources of degradation that arise when attempting to quantize graphs and, presented `Degree-Quant`, an architecture-agnostic and stable method for training quantized GNN models that can be accelerated using off-the-shelf hardware. With proposed framework, inaccurate weights updates are reduced by keeping high-degree nodes at full precision more often and, the rounding error is controlled with a percentiles tracking mechanism. We validate our method on six datasets and show, unlike previous quantization attempts, that models generalize to unseen graphs. With 4-bit weights and activations we achieve $8\times$ compression while surpassing strong baselines by margins regularly exceeding 20%. At 8-bits, models trained with DQ perform on par or better than the baselines while achieving up to 4.7$\times$ lower latency than FP32 models. Our work offers a comprehensive foundation for future work in this area and is a first step towards enabling GNNs to be deployed more widely, including to resource constrained devices such as smartphones.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

After a decade of accelerated progress in the different areas of machine learning (ML), it has become virtually impossible to abstain from using of ML-powered applications and services in our everyday lives. Recommended systems, computational photography, speech translation or, getting directions are only a handful of examples of what can be done from our smartphones or other battery-powered devices around us. However, despite the known benefits of running such applications on-device (e.g. privacy, works offline, real-time potential), the increasing complexity of such systems and their reliance on millions, or even billions, of parameters and complex execution patterns, have prevented these from leaving the Cloud, where inference takes place. As a result, only a fraction of such applications remains lightweight enough (e.g. keyword spotting, next-word prediction) to run on-device. *What has to change to make more complex and better performing models run efficiently on constrained devices?*

This thesis has studied such challenging problem from three different perspectives: first, by identifying a new paradigm for compression frameworks by which model acceleration is achieved by alleviating the impact of data movement; second, by solving the inherent problem of numerical degradation that has prevented the adoption and deployment of fast Winograd convolutions for models making use of integer arithmetic; and third, by studying mechanisms that enable the learning of quantized graph neural networks and that overcome two of the limitations that do not appear in other mainstream forms of machine learning. Each of this three contributions remain orthogonal to other techniques in the literature. For example, on-the-fly models benefit from structural pruning methods [400, 257] to achieve further speedups. Similarly, quantized GNNs would benefit from other techniques such as sampling [138, 405] to perform inference on even larger graphs. With regards to Winograd-

aware layers, these can incorporate the changes suggested in [169] and enable more striding options, which might be required when considering larger inputs or video streams.

Each of these three contributions offer an answer to a different, concrete and, original research question instead of providing an incremental solution to a set of well studied problems. The remaining of this section revisits such questions and, provides a concise summary on how these were addressed.

In Chapter 3 we addressed the question of *how would a compression framework that achieves acceleration by generating the model parameters on-the-fly instead of retrieving them from off-chip memory would look like?* We began by providing a definition for on-the-fly compression frameworks, making their reliance on an inference-conscious decompression stage one of their core components. We investigated the use of OVSF codes for image classification and keyword spotting applications and, show these codes are suitable candidates for on-the-fly models thanks to their mutually orthogonal, binary and, deterministic properties. With `unzipFPGA` we derived and mapped high-performing CNNs on FPGAs making use of `CNN-WGen`, a novel and custom hardware weights generator for OVSF models. Our implementation of on-the-fly models surpassed structurally pruned baselines in terms of throughput under several settings, while showing the largest margins in reduced $1\times$ and $2\times$ bandwith configurations.

In Chapter 4 we considered the long standing problem of numerical degradation inherent to Winograd convolutions for deployments making use of quantization. Concretely, the questions we provided an answer for was: *Can we capture the numerical errors at training time, introduce relaxations in the form of the transformation matrices and, achieve faster inference than other algorithms such as* `im2row`*?* We presented a Winograd-aware formulation for convolutional layers that exposed the numerical degradation introduced by quantization and the use of large tiles at training time. Those layers defined the search space of `wiNAS`, a framework that jointly optimized a given macro-architecture in terms of latency and accuracy, which proved useful to deriving quantized CNNs that are over $1.5\times$ faster on mobile CPUs than when making use of highly optimized `im2row` convolutions.

Lastly, we considered in Chapter 5 the problem of optimizing inference for GNNs, an emerging form of ML that operates on unstructured data representations, through quantization. We asked the question: *What are the unique challenges that make GNNs particularly difficult to accelerate using quantization?* We identified that the aggregation stage is of particular concern since the mean and variance of aggregated messages can dramatically vary depending on both the node degree and the type of layer. When this happens, the range of values to represent under a fixed bitwidth could be large, leaving the majority of outputs

from the aggregation stage quantized at low numerical resolutions. To address, this we introduced `Degree-Quant`, a stochastic topology-aware re-formulation of quantization-aware training suitable for graph. With `Degree-Quant`, INT8 models perform as well as FP32 models in most cases, INT4 models are up to 26% better than other QAT baselines. In addition, we show that INT8 quantization enables up to $4.7\times$ speedups on CPU.

6.2 Future Work

As it was introduced in the first pages of this thesis, the fields of efficient ML and on-device ML have been steadily evolving over the last few years. The contributions of this thesis to these fields have focused on developing novel optimization techniques for inference involving compression, algorithms for convolutions, neural architecture search and, quantization. These contributions have been evaluated under supervised training paradigms and, primarily for classification tasks, enabling direct comparisons with other works in the literature. In addition to the extensions and future work ideas discussed at the end of each chapter, the following research directions would further advance the field and, enable the deployment of better performing ML-powered applications on constrained devices.

Training on-device. Until recently, the vast majority of applications running ML models on-device have first been trained on the cloud using a vast and centralized dataset, often containing data from end-users (e.g. the type of products the user buys, search queries on an e-commerce site or, photos uploaded to a on-line backup service). Training on these large datasets, which can be pre-processed and curated for different tasks, often results in high quality models that can later on be integrated into products and applications that run on-device (e.g. a smartphone, smart speaker, self-driving cars, etc) where inference, or a portion of it, takes place. However, in certain settings data might be too sensible to be sent to the cloud, where users have little to no control over it. In such cases, shifting from centralized to a Federated Learning (FL) setup might be necessary. FL is an emerging form of machine learning where nodes are commodity devices such as smartphones, wearables or other edge/IoT devices. The task is to collaboratively train a *global model* while each participating node exclusively uses its own data. There is a server that orchestrates the FL training process, where all models in a given FL round get aggregated and a new global model is created. A substantial amount of works have explored ways of reducing the communication costs [195] and propose aggregation strategies [247, 292] to achieve faster convergence. However, addressing the challenge of how to train models efficiently

has not attracted so much attention in the context of FL with only a few works addressing this through distillation [226] or model scaling [163]. This challenge limits the suitability of FL for more compute and memory demanding tasks due to devices being several orders of magnitude more constrained that then type of hardware found in modern datacenters. Taking this into consideration, what are the optimization techniques that can be used at training time that would make training more efficient while still result in a good performing global model? Do existing techniques need to be adapted to the FL setting given that data in the clients is often scarce, unbalanced (i.e. non-IID) and, above all private?

Approximate computations. Among the core techniques used to accelerate computations and lower the memory consumption of multiple ML workloads, the use of quantization and sparsity stand out. These techniques, which have ramified over the years as shown in Section 2.1, can collectively be seen as methods to *approximate* operands for functions (e.g. a convolutional layer or a dot product) that enable their processing to be more lightweight. However, the core functions used in ML models have remained intact. For example, in convolutional layers, regardless of the approximation level of its operands, they still perform an exact discrete convolution. One immediate approach to approximate an operation would be to first project its operands to a lower-dimensional space, perform the exact computation and, project the output back to the original space. A number of works following this framing have been proposed aiming at approximating matrix multiplications (AMM) [38, 116, 345]. Such approximations can often be parameterized for either numerical precision or speed, effectively becoming yet another orthogonal dimension to consider when ML workloads, which heavily rely on `matmul`(s), need to be accelerated. Another form of approximation are AdderNets [52, 222] which aim at entirely replacing multiplications with additions, which could also translate into additional benefits in terms of energy consumption and, chip area. Approximating the operator and not just the operands could be one of the key ingredients to make on-device training, as in FL, more viable.

Larger models on the horizon. Supervised learning has been the form of training that sustained most of the advancements done in the last decade, with the 1.3M labeled images of the ImageNet dataset being the golden standard for image classification. Other, but not many, large-scale annotated datasets exist for various applications [227, 18, 147]. However, scaling to larger datasets becomes an arduous tasks only attainable by large industry players, with Google's JFT-300M [313] and Facebook's IG-1B-Targeted [384], each containing 300M and 1Billion images with fuzzy labels, as examples. These datasets are over two orders of magnitude larger than ImageNet but have not been released to the research

community. To circumvent the challenge of collecting and annotating large datasets, the community has instead shifted to an alternative training paradigm which learning is done in an unsupervised manner, i.e. without labels. Throughout this process or at different stages in a multi-stage training pipeline, outputs from the model can be treated as labels (i.e self-supervised training) and, some labeled inputs can be included (i.e. semi-supervised training). These forms of learning benefits from much larger models than supervised training does [55], therefore it would be reasonable to expect that the resulting models for a downstream task would remain large. For example, a common benchmarking architecture is ResNet-50 ($4\times$), which is a standard ResNet-50 but with $4\times$ more channels in each layer, effectively quadrupling the memory footprint needed for activations at training time and, to update its 100M parameters. Unsupervised learning could play a decisive role in a future where ML becomes ubiquitous and leverages large amounts of multisensorial unlabeled data. However, for that to be realizable, the existing training methodologies, architectures and, unsupervised data collection routines would need to be adapted to the on-device setting. This is because such vasts amounts of data would become intractable for the datacenters to process and for the networking infrastructure to transfer, leaving the on-device processing and training as the only apparent option to learn from this rich yet unlabeled data.

References

[1] A graph placement methodology for fast chip design. *Nature 594*, 7862 (2021), 207–212.

[2] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] ABDELFATTAH, M. S., DUDZIAK, Ł., CHAU, T., LEE, R., KIM, H., AND LANE, N. D. Best of Both Worlds: AutoML Codesign of a CNN and its Hardware Accelerator. In *Design Automation Conference (DAC)* (2020).

[4] ABDELFATTAH, M. S., HAN, D., BITAR, A., DICECCO, R., O'CONNELL, S., SHANKER, N., CHU, J., PRINS, I., FENDER, J., LING, A. C., AND CHIU, G. R. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), pp. 411–4117.

[5] ABDELFATTAH, M. S., MEHROTRA, A., DUDZIAK, Ł., AND LANE, N. D. Zero-cost proxies for lightweight {nas}. In *International Conference on Learning Representations* (2021).

[6] ABTAHI, T., SHEA, C., KULKARNI, A., AND MOHSENIN, T. Accelerating convolutional neural network with fft on embedded hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26*, 9 (Sep. 2018), 1737–1749.

[7] ACHANTA, R., SHAJI, A., SMITH, K., LUCCHI, A., FUA, P., AND SÜSSTRUNK, S. Slic superpixels compared to state-of-the-art superpixel methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence 34*, 11 (2012), 2274–2282.

[8] ADACHI, F., ET AL. Wideband ds-cdma for next-generation mobile communications systems. *IEEE communications Magazine 36*, 9 (1998), 56–69.

[9] ADAP. Flower. https://flower.dev/conf/flower-summit-2021. Accessed: 2021-07-31.

[10] ALISTARH, D., GRUBIC, D., LI, J., TOMIOKA, R., AND VOJNOVIC, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[11] ALIZADEH, M., FERNÁNDEZ-MARQUÉS, J., LANE, N. D., AND GAL, Y. A systematic study of binary neural networks' optimisation. In *International Conference on Learning Representations* (2019).

[12] Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings* (2018), OpenReview.net.

[13] Alwani, M., Chen, H., Ferdman, M., and Milder, P. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), pp. 1–12.

[14] Anderson, A., and Gregg, D. Optimal dnn primitive selection with partitioned boolean quadratic programming. *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018* (2018).

[15] Andreev, B. D., Titlebaum, E. L., and Friedman, E. G. Orthogonal code generator for 3g wireless transceivers. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI* (New York, NY, USA, 2003), GLSVLSI '03, ACM, pp. 229–232.

[16] Aojun Zhou, Anbang Yao, Y. G. L. X., and Chen, Y. Incremental network quantization: Towards lossless cnns with low-precision weights. In *International Conference on Learning Representations,ICLR2017* (2017).

[17] Arai, J., Shiokawa, H., Yamamuro, T., Onizuka, M., and Iwamura, S. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016), pp. 22–31.

[18] Ardila, R., Branson, M., Davis, K., Henretty, M., Kohler, M., Meyer, J., Morais, R., Saunders, L., Tyers, F. M., and Weber, G. Common voice: A massively-multilingual speech corpus. In *Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020)* (2020), pp. 4211–4215.

[19] Arm Software. Arm compute library. https://arm-software.github.io/ComputeLibrary/latest/index.xhtml. Accessed: 2021-05-15.

[20] Assael, Y. M., Shillingford, B., Whiteson, S., and de Freitas, N. Lipnet: End-to-end sentence-level lipreading, 2016.

[21] Auten, A., Tomei, M., and Kumar, R. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), pp. 1–6.

[22] Aydonat, U., O'Connell, S., Capalija, D., Ling, A. C., and Chiu, G. R. An OpenCL™ Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (2017), p. 55–64.

[23] Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., and Blundell, C. Agent57: Outperforming the Atari human benchmark. In *Proceedings of the 37th International Conference on Machine Learning* (13–18 Jul 2020), H. D. III and A. Singh, Eds., vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 507–517.

[24] Baek, M., DiMaio, F., Anishchenko, I., Dauparas, J., Ovchinnikov, S., Lee, G. R., Wang, J., Cong, Q., Kinch, L. N., Schaeffer, R. D., Millán, C., Park, H., Adams, C., Glassman, C. R., DeGiovanni, A., Pereira, J. H., Rodrigues, A. V., van Dijk, A. A., Ebrecht, A. C., Opperman, D. J., Sagmeister, T., Buhlheller, C., Pavkov-Keller, T., Rathinaswamy, M. K., Dalwadi, U., Yip, C. K., Burke, J. E., Garcia, K. C., Grishin, N. V., Adams, P. D., Read, R. J., and Baker, D. Accurate prediction of protein structures and interactions using a three-track neural network. *Science* (2021).

[25] Bahri, M., Bahl, G., and Zafeiriou, S. Binary graph neural networks, 2021.

[26] BALAJI, V., AND LUCIA, B. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)* (2018), pp. 203–214.

[27] BANBURY, C., ZHOU, C., FEDOROV, I., NAVARRO, R. M., THAKKER, U., GOPE, D., REDDI, V. J., MATTINA, M., AND WHATMOUGH, P. N. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers, 2021.

[28] BANNER, R., NAHSHAN, Y., AND SOUDRY, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.

[29] BARABASZ, B. Quantaized winograd/toom-cook convolution for dnns: Beyond canonical polynomials base, 2020.

[30] BARABASZ, B., ANDERSON, A., SOODHALTER, K. M., AND GREGG, D. Error analysis and improving the accuracy of winograd convolution for deep neural networks, 2019.

[31] BARABASZ, B., AND GREGG, D. Winograd convolution for dnns: Beyond linear polynomials, 2019.

[32] BENDER, G., KINDERMANS, P.-J., ZOPH, B., VASUDEVAN, V., AND LE, Q. Understanding and simplifying one-shot architecture search. In *Proceedings of the 35th International Conference on Machine Learning* (Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 550–559.

[33] BENGIO, Y., LÉONARD, N., AND COURVILLE, A. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[34] BEUTEL, D. J., TOPAL, T., MATHUR, A., QIU, X., PARCOLLET, T., DE GUSMÃO, P. P. B., AND LANE, N. D. Flower: A friendly federated learning research framework, 2020.

[35] BHATTACHARYA, S., AND LANE, N. D. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (New York, NY, USA, 2016), SenSys '16, Association for Computing Machinery, p. 176–189.

[36] BIANCHI, F., GRATTAROLA, D., LIVI, L., AND ALIPPI, C. Graph neural networks with convolutional arma filters. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 01 (jan 5555), 1–1.

[37] BLAHUT, R. E. *Fast Algorithms for Signal Processing*. Cambridge University Press, 2010.

[38] BLALOCK, D., AND GUTTAG, J. Multiplying matrices without multiplying. In *Proceedings of the 38th International Conference on Machine Learning* (18–24 Jul 2021), M. Meila and T. Zhang, Eds., vol. 139 of *Proceedings of Machine Learning Research*, PMLR, pp. 992–1004.

[39] BLALOCK, D., ORTIZ, J. J. G., FRANKLE, J., AND GUTTAG, J. What is the state of neural network pruning?, 2020.

[40] BRESSON, X., AND LAURENT, T. Residual gated graph convnets, 2018.

[41] BRILLET, L. F., MANCINI, S., CLEYET-MERLE, S., AND NICOLAS, M. Tunable cnn compression through dimensionality reduction. In *2019 IEEE International Conference on Image Processing (ICIP)* (2019), pp. 3851–3855.

[42] BROCK, A., LIM, T., RITCHIE, J., AND WESTON, N. SMASH: One-shot model architecture search through hypernetworks. In *International Conference on Learning Representations* (2018).

[43] BROWN, T., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J. D., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., AGARWAL, S., HERBERT-VOSS, A., KRUEGER, G., HENIGHAN, T., CHILD, R., RAMESH, A., ZIEGLER, D., WU, J., WINTER, C., HESSE, C., CHEN, M., SIGLER, E., LITWIN, M., GRAY, S., CHESS, B., CLARK, J., BERNER, C., MCCANDLISH, S., RADFORD, A., SUTSKEVER, I., AND AMODEI, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 1877–1901.

[44] BRUNA, J., ZAREMBA, W., SZLAM, A., AND LECUN, Y. Spectral networks and locally connected networks on graphs, 2014.

[45] BULAT, A., AND TZIMIROPOULOS, G. Xnor-net++: Improved binary neural networks, 2019.

[46] CAI, H., GAN, C., WANG, T., ZHANG, Z., AND HAN, S. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations* (2020).

[47] CAI, H., ZHU, L., AND HAN, S. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations* (2019).

[48] CARREIRA-PERPINAN, M. A., AND IDELBAYEV, Y. ”learning-compression” algorithms for neural net pruning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 8532–8541.

[49] CASAS, S., GULINO, C., LIAO, R., AND URTASUN, R. Spagnn: Spatially-aware graph neural networks for relational behavior forecasting from sensor data. In *2020 IEEE International Conference on Robotics and Automation (ICRA)* (2020), pp. 9491–9497.

[50] CASAS, S., LUO, W., AND URTASUN, R. Intentnet: Learning to predict intention from raw sensor data. In *Proceedings of The 2nd Conference on Robot Learning* (29–31 Oct 2018), A. Billard, A. Dragan, J. Peters, and J. Morimoto, Eds., vol. 87 of *Proceedings of Machine Learning Research*, PMLR, pp. 947–956.

[51] CHANG, A. X. M., ZAIDY, A., GOKHALE, V., AND CULURCIELLO, E. Compiling Deep Learning Models for Custom Hardware Accelerators. *arXiv preprint arXiv:1708.00117* (2017).

[52] CHEN, H., WANG, Y., XU, C., SHI, B., XU, C., TIAN, Q., AND XU, C. Addernet: Do we really need multiplications in deep learning? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2020).

[53] CHEN, J., MA, T., AND XIAO, C. Fastgcn: Fast learning with graph convolutional networks via importance sampling, 2018.

[54] CHEN, J., ZHU, J., AND SONG, L. Stochastic training of graph convolutional networks with variance reduction, 2018.

[55] CHEN, T., KORNBLITH, S., NOROUZI, M., AND HINTON, G. A simple framework for contrastive learning of visual representations. In *Proceedings of the 37th International Conference on Machine Learning* (13–18 Jul 2020), H. D. III and A. Singh, Eds., vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 1597–1607.

[56] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[57] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 578–594.

[58] Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost, 2016.

[59] Chen, X., Wang, Y., Xie, X., Hu, X., Basak, A., Liang, L., Yan, M., Deng, L., Ding, Y., Du, Z., Chen, Y., and Xie, Y. Rubik: A hierarchical architecture for efficient graph learning, 2020.

[60] Chen, Y., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (JSSC) 52*, 1 (2017), 127–138.

[61] Cheng, X., Rao, Z., Chen, Y., and Zhang, Q. Explaining knowledge distillation by quantifying the knowledge. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2020).

[62] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR abs/1410.0759* (2014).

[63] Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C.-J. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (New York, NY, USA, 2019), KDD '19, Association for Computing Machinery, p. 257–266.

[64] Chiu, C.-C., Sainath, T. N., Wu, Y., Prabhavalkar, R., Nguyen, P., Chen, Z., Kannan, A., Weiss, R. J., Rao, K., Gonina, E., et al. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2018), IEEE, pp. 4774–4778.

[65] Choi, Y., El-Khamy, M., and Lee, J. Compression of deep convolutional neural networks under joint sparsity constraints, 2018.

[66] Choi, Y., El-Khamy, M., and Lee, J. Universal deep neural network compression. *IEEE Journal of Selected Topics in Signal Processing 14*, 4 (2020), 715–726.

[67] Choukroun, Y., Kravchik, E., Yang, F., and Kisilev, P. Low-bit quantization of neural networks for efficient inference, 2019.

[68] Chowdhery, A., Warden, P., Shlens, J., Howard, A., and Rhodes, R. Visual wake words dataset, 2019.

[69] Cong, J., and Xiao, B. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning – ICANN 2014* (Cham, 2014), S. Wermter, C. Weber, W. Duch, T. Honkela, P. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, Eds., Springer International Publishing, pp. 281–290.

[70] Corso, G., Cavalleri, L., Beaini, D., Liò, P., and Veličković, P. Principal neighbourhood aggregation for graph nets. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 13260–13271.

[71] Courbariaux, M., Bengio, Y., and David, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations, 2015.

[72] COURBARIAUX, M., BENGIO, Y., AND DAVID, J.-P. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 3123–3131.

[73] DAVID PEARCE, C. D. Speech processing, transmission and quality aspects (stq); distributed speech recognition; front-end feature extraction algorithm; compression algorithms.

[74] DEFFERRARD, M., BRESSON, X., AND VANDERGHEYNST, P. Convolutional neural networks on graphs with fast localized spectral filtering, 2017.

[75] DENG, J., DONG, W., SOCHER, R., JIA LI, L., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *CVPR* (2009).

[76] DENG, X., AND ZHANG, Z. An embarrassingly simple approach to training ternary weight networks, 2020.

[77] DENTON, E., ZAREMBA, W., BRUNA, J., LECUN, Y., AND FERGUS, R. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, p. 1269–1277.

[78] DERR, T., MA, Y., AND TANG, J. Signed graph convolutional networks. In *2018 IEEE International Conference on Data Mining (ICDM)* (2018), pp. 929–934.

[79] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (Minneapolis, Minnesota, June 2019), Association for Computational Linguistics, pp. 4171–4186.

[80] DHILLON, I. S., GUAN, Y., AND KULIS, B. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence 29*, 11 (2007), 1944–1957.

[81] DOLLÁR, P., APPEL, R., BELONGIE, S., AND PERONA, P. Fast feature pyramids for object detection. *PAMI* (2014).

[82] DOLLÁR, P., BELONGIE, S., AND PERONA, P. The fastest pedestrian detector in the west. In *BMVC* (2010).

[83] DOLLÁR, P., WELINDER, P., AND PERONA, P. Cascaded pose regression. In *CVPR* (2010).

[84] DONG, J.-D., CHENG, A.-C., JUAN, D.-C., WEI, W., AND SUN, M. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *Proceedings of the European Conference on Computer Vision (ECCV)* (September 2018).

[85] DONG, X., THANOU, D., RABBAT, M., AND FROSSARD, P. Learning graphs from data: A signal representation perspective. *IEEE Signal Processing Magazine 36*, 3 (2019), 44–63.

[86] DONG, X., AND YANG, Y. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations* (2020).

[87] DONG, Y., NI, R., LI, J., CHEN, Y., ZHU, J., AND SU, H. Learning accurate low-bit deep neural networks with stochastic quantization, 2017.

[88] DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., GELLY, S., USZKOREIT, J., AND HOULSBY, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations* (2021).

[89] DU, Z., FASTHUBER, R., CHEN, T., IENNE, P., LI, L., LUO, T., FENG, X., CHEN, Y., AND TEMAM, O. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (2015), pp. 92–104.

[90] DUDZIAK, L., CHAU, T., ABDELFATTAH, M., LEE, R., KIM, H., AND LANE, N. Brp-nas: Prediction-based nas using gcns. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 10480–10490.

[91] DUHAMEL, P. Implementation of "split-radix" fft algorithms for complex, real, and real-symmetric data. *IEEE Transactions on Acoustics, Speech, and Signal Processing 34*, 2 (1986), 285–295.

[92] DUKHAN, M. The indirect convolution algorithm, 2019.

[93] DUONG, C. N., LUU, K., QUACH, K. G., AND LE, N. Shrinkteanet: Million-scale lightweight face recognition via shrinking teacher-student networks, 2019.

[94] DUVENAUD, D. K., MACLAURIN, D., IPARRAGUIRRE, J., BOMBARELL, R., HIRZEL, T., ASPURU-GUZIK, A., AND ADAMS, R. P. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems* (2015), pp. 2224–2232.

[95] DWIVEDI, V. P., JOSHI, C. K., LAURENT, T., BENGIO, Y., AND BRESSON, X. Benchmarking graph neural networks, 2020.

[96] ELSEN, E., DUKHAN, M., GALE, T., AND SIMONYAN, K. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2020).

[97] ESSER, S. K., MCKINSTRY, J. L., BABLANI, D., APPUSWAMY, R., AND MODHA, D. S. Learned step size quantization. In *International Conference on Learning Representations* (2020).

[98] F. ADACHI, M. SAWAHASHI, K. O. Tree-structured generation of orthogonal spreading codes with different lengths for forward link of ds-cdma mobile radio. *Electronics Letters 33* (January 1997), 27–28(1).

[99] FEDOROV, I., ADAMS, R. P., MATTINA, M., AND WHATMOUGH, P. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers. In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.

[100] FEDOROV, I., ADAMS, R. P., MATTINA, M., AND WHATMOUGH, P. N. SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers. In *Proceedings of the Neural Information Processing Systems (NeurIPS) Conference 2019* (2019).

[101] FENG, B., WANG, Y., LI, X., YANG, S., PENG, X., AND DING, Y. Sgquant: Squeezing the last bit on graph neural networks with specialized quantization, 2020.

[102] FERNÁNDEZ-MARQUÉS, J., TSENG, V. W.-S., BHATTACHARA, S., AND LANE, N. D. On-the-Fly Deterministic Binary Filters for Memory Efficient Keyword Spotting Applications on Embedded Devices. In *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning (EMDL)* (2018), ACM, p. 13–18.

[103] FERNÁNDEZ-MARQUÉS, J., VINCENT, W.-S. T., BHATTACHARA, S., AND LANE, N. D. BinaryCmd: Keyword Spotting with deterministic binary basis. In *Conference on Machine Learning and Systems (MLSys)* (2018).

[104] FERNANDEZ-MARQUES, J., WHATMOUGH, P., MUNDY, A., AND MATTINA, M. Searching for winograd-aware quantized networks. In *Proceedings of Machine Learning and Systems* (2020), I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, pp. 14–29.

[105] FEY, M., AND LENSSEN, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019).

[106] FOWERS, J., OVTCHAROV, K., PAPAMICHAEL, M., MASSENGILL, T., LIU, M., LO, D., ALKALAY, S., HASELMAN, M., ADAMS, L., GHANDI, M., HEIL, S., PATEL, P., SAPEK, A., WEISZ, G., WOODS, L., LANKA, S., REINHARDT, S. K., CAULFIELD, A. M., CHUNG, E. S., AND BURGER, D. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), pp. 1–14.

[107] FRANKLE, J., AND CARBIN, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations* (2019).

[108] GAO, H., WANG, Z., AND JI, S. Channelnets: Compact and efficient convolutional neural networks via channel-wise convolutions. In *Advances in Neural Information Processing Systems* (2018), S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc.

[109] GAO, M., WANG, Y., AND WAN, L. Residual error based knowledge distillation. *Neurocomputing 433* (2021), 154–161.

[110] GAO, Y., PARCOLLET, T., ZAIEM, S., FERNANDEZ-MARQUES, J., DE GUSMAO, P. P. B., BEUTEL, D. J., AND LANE, N. D. End-to-end speech recognition from federated acoustic models, 2021.

[111] GAO, Y., YANG, H., ZHANG, P., ZHOU, C., AND HU, Y. Graph neural architecture search. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20* (7 2020), C. Bessiere, Ed., International Joint Conferences on Artificial Intelligence Organization, pp. 1403–1409. Main track.

[112] GARG, I., PANDA, P., AND ROY, K. A low effort approach to structured cnn design using pca. *IEEE Access 8* (2020), 1347–1360.

[113] GE, S., LUO, Z., ZHAO, S., JIN, X., AND ZHANG, X.-Y. Compressing deep neural networks for efficient visual inference. In *2017 IEEE International Conference on Multimedia and Expo (ICME)* (2017), pp. 667–672.

[114] GEIGER, L., AND TEAM, P. Larq: An open-source library for training binarized neural networks. *Journal of Open Source Software 5*, 45 (Jan. 2020), 1746.

[115] GENG, T., LI, A., SHI, R., WU, C., WANG, T., LI, Y., HAGHI, P., TUMEO, A., CHE, S., REINHARDT, S., AND HERBORDT, M. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing, 2020.

[116] GHASHAMI, M., DESAI, A., AND PHILLIPS, J. M. Improved practical matrix sketching with guarantees. In *Algorithms - ESA 2014* (Berlin, Heidelberg, 2014), A. S. Schulz and D. Wagner, Eds., Springer Berlin Heidelberg, pp. 467–479.

[117] GHOLAMI, A., KIM, S., DONG, Z., YAO, Z., MAHONEY, M. W., AND KEUTZER, K. A survey of quantization methods for efficient neural network inference, 2021.

[118] GILMER, J., SCHOENHOLZ, S. S., RILEY, P. F., VINYALS, O., AND DAHL, G. E. Neural message passing for quantum chemistry. *CoRR abs/1704.01212* (2017).

[119] GLASSEY, C. R., AND KARP, R. M. On the optimality of huffman trees. *SIAM Journal on Applied Mathematics 31*, 2 (1976), 368–378.

[120] GOKHALE, V., ZAIDY, A., CHANG, A. X. M., AND CULURCIELLO, E. Snowflake: An Efficient Hardware Accelerator for Convolutional Neural Networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)* (2017), pp. 1–4.

[121] GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23*, 1 (Mar. 1991), 5–48.

[122] GONG, C., JIANG, Z., WANG, D., LIN, Y., LIU, Q., AND PAN, D. Z. Mixed precision neural architecture search for energy efficient deep learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2019), pp. 1–7.

[123] GONG, R., LIU, X., JIANG, S., LI, T., HU, P., LIN, J., YU, F., AND YAN, J. Differentiable soft quantization: Bridging full-precision and low-bit neural networks, 2019.

[124] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, Oct. 2012), USENIX Association, pp. 17–30.

[125] GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. Generative adversarial nets. In *Advances in Neural Information Processing Systems* (2014), Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27, Curran Associates, Inc.

[126] GOOGLE LLC. Tensorflow lite. https://www.tensorflow.org/lite. Accessed: 2021-07-10.

[127] GOPE, D., BEU, J., THAKKER, U., AND MATTINA, M. Ternary mobilenets via per-layer hybrid filter banks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (June 2020).

[128] GORDON, A., EBAN, E., NACHUM, O., CHEN, B., WU, H., YANG, T.-J., AND CHOI, E. Morphnet: Fast simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2018).

[129] GOU, J., YU, B., MAYBANK, S. J., AND TAO, D. Knowledge distillation: A survey. *International Journal of Computer Vision 129*, 6 (Mar 2021), 1789–1819.

[130] GRAHAM, B., EL-NOUBY, A., TOUVRON, H., STOCK, P., JOULIN, A., JÉGOU, H., AND DOUZE, M. Levit: a vision transformer in convnet's clothing for faster inference, 2021.

[131] GU, J., LIU, Y., GAO, Y., AND ZHU, M. Opencl caffe: Accelerating and enabling a cross platform machine learning framework. In *Proceedings of the 4th International Workshop on OpenCL* (New York, NY, USA, 2016), IWOCL '16, ACM, pp. 8:1–8:5.

[132] GUAN, Y., LIANG, H., XU, N., WANG, W., SHI, S., CHEN, X., SUN, G., ZHANG, W., AND CONG, J. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2017), pp. 152–159.

[133] GUI, C., ZHENG, L., HE, B., LIU, C., CHEN, X., LIAO, X., AND JIN, H. A survey on graph processing accelerators: Challenges and opportunities, 2019.

[134] GUO, K., SUI, L., QIU, J., YU, J., WANG, J., YAO, S., HAN, S., WANG, Y., AND YANG, H. Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 37*, 1 (2018), 35–47.

[135] GUO, Y., YAO, A., AND CHEN, Y. Dynamic network surgery for efficient dnns. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2016), NIPS'16, Curran Associates Inc., p. 1387–1395.

[136] HA, D., DAI, A., AND LE, Q. V. HyperNetworks. In *International Conference on Learning Representations (ICLR)* (2017).

[137] HAMILTON, W. L., YING, R., AND LESKOVEC, J. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017), NIPS'17, Curran Associates Inc., p. 1025–1035.

[138] HAMILTON, W. L., YING, R., AND LESKOVEC, J. Inductive representation learning on large graphs, 2018.

[139] HAN, S., LIU, X., MAO, H., PU, J., PEDRAM, A., HOROWITZ, M. A., AND DALLY, W. J. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), ISCA '16, IEEE Press, p. 243–254.

[140] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

[141] HAN, S., POOL, J., TRAN, J., AND DALLY, W. J. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Cambridge, MA, USA, 2015), NIPS'15, MIT Press, p. 1135–1143.

[142] HARIHARAN, B., ZITNICK, C. L., AND DOLLÁR, P. Detecting objects using deformation dictionaries. In *CVPR* (2014).

[143] HASSIBI, B., AND STORK, D. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems* (1993), S. Hanson, J. Cowan, and C. Giles, Eds., vol. 5, Morgan-Kaufmann.

[144] HE, B., GOVINDARAJU, N. K., LUO, Q., AND SMITH, B. Efficient gather and scatter operations on graphics processors. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (2007), pp. 1–12.

[145] HE, K., ZHANG, X., REN, S., AND SUN, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (December 2015).

[146] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun 2016).

[147] HE, R., AND MCAULEY, J. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web* (Republic and Canton of Geneva, CHE, 2016), WWW '16, International World Wide Web Conferences Steering Committee, p. 507–517.

[148] He, Y., Kang, G., Dong, X., Fu, Y., and Yang, Y. Soft filter pruning for accelerating deep convolutional neural networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (2018), IJCAI'18, AAAI Press, p. 2234–2240.

[149] He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision (ECCV)* (2018).

[150] He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. *Lecture Notes in Computer Science* (2018), 815–832.

[151] He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *The IEEE International Conference on Computer Vision (ICCV)* (Oct 2017).

[152] Henaff, M., Bruna, J., and LeCun, Y. Deep convolutional networks on graph-structured data, 2015.

[153] Hernández-Lobato, J. M., Gelbart, M. A., Reagen, B., Adolf, R., Hernández-Lobato, D., Whatmough, P. N., Brooks, D., Wei, G.-Y., and Adams, R. P. Designing neural network hardware accelerators with decoupled objective evaluations. In *NIPS workshop on Bayesian Optimization* (2016).

[154] Higham, N. J. *Accuracy and Stability of Numerical Algorithms*, second ed. Society for Industrial and Applied Mathematics, 2002.

[155] Highlander, T., and Rodriguez, A. Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add. *Procedings of the British Machine Vision Conference 2015* (2015).

[156] Hill, A. P., Prince, P., Piña Covarrubias, E., Doncaster, C. P., Snaddon, J. L., and Rogers, A. Audiomoth: Evaluation of a smart open acoustic device for monitoring biodiversity and the environment. *Methods in Ecology and Evolution 9*, 5 (2018), 1199–1211.

[157] Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop* (2015).

[158] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[159] Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, 2021.

[160] Hong, C., Sukumaran-Rajam, A., Bandyopadhyay, B., Kim, J., Kurt, S. E., Nisa, I., Sabhlok, S., Çatalyürek, U. V., Parthasarathy, S., and Sadayappan, P. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2018), HPDC '18, Association for Computing Machinery, p. 66–79.

[161] Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., and Sadayappan, P. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2019), PPoPP '19, Association for Computing Machinery, p. 300–314.

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

[163] HORVATH, S., LASKARIDIS, S., ALMEIDA, M., LEONTIADIS, I., VENIERIS, S. I., AND LANE, N. D. Fjord: Fair and accurate federated learning under heterogeneous targets with ordered dropout, 2021.

[164] HOWARD, A., SANDLER, M., CHU, G., CHEN, L.-C., CHEN, B., TAN, M., WANG, W., ZHU, Y., PANG, R., VASUDEVAN, V., LE, Q. V., AND ADAM, H. Searching for mobilenetv3, 2019.

[165] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[166] HSU, C.-H., CHANG, S.-H., LIANG, J.-H., CHOU, H.-P., LIU, C.-H., CHANG, S.-C., PAN, J.-Y., CHEN, Y.-T., WEI, W., AND JUAN, D.-C. Monas: Multi-objective neural architecture search using reinforcement learning, 2018.

[167] HU, H., PENG, R., TAI, Y.-W., AND TANG, C.-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures, 2016.

[168] HU, Q., YANG, B., XIE, L., ROSA, S., GUO, Y., WANG, Z., TRIGONI, N., AND MARKHAM, A. Randla-net: Efficient semantic segmentation of large-scale point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2020).

[169] HUANG, D., ZHANG, X., ZHANG, R., ZHI, T., HE, D., GUO, J., LIU, C., GUO, Q., DU, Z., LIU, S., CHEN, T., AND CHEN, Y. Dwm: A decomposable winograd method for convolution acceleration. *Proceedings of the AAAI Conference on Artificial Intelligence 34*, 04 (Apr. 2020), 4174–4181.

[170] HUANG, G., LIU, Z., VAN DER MAATEN, L., AND WEINBERGER, K. Q. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (July 2017).

[171] HUANG, W., ZHANG, T., RONG, Y., AND HUANG, J. Adaptive sampling towards fast graph representation learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2018), NIPS'18, Curran Associates Inc., p. 4563–4572.

[172] HUNTER, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering 9*, 3 (2007), 90–95.

[173] IANDOLA, F. N., HAN, S., MOSKEWICZ, M. W., ASHRAF, K., DALLY, W. J., AND KEUTZER, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[174] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[175] JACOB, B., KLIGYS, S., CHEN, B., ZHU, M., TANG, M., HOWARD, A., ADAM, H., AND KALENICHENKO, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Jun 2018).

[176] JAIN, S. R., GURAL, A., WU, M., AND DICK, C. H. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks, 2020.

[177] JANOWSKY, S. A. Pruning versus clipping in neural networks. *Phys. Rev. A 39* (Jun 1989), 6600–6603.

[178] JETTE, M. A., YOO, A. B., AND GRONDONA, M. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[179] JIA, Y. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[180] JIA, Z., LIN, S., GAO, M., ZAHARIA, M., AND AIKEN, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of Machine Learning and Systems 2020*. 2020, pp. 187–198.

[181] JIAN-HAO LUO, J. W., AND LIN, W. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. In *International Conference on Computer Vision (ICCV)* (October 2017).

[182] JIANG, J., CHEN, X., AND TSUI, C. A reconfigurable winograd CNN accelerator with nesting decomposition algorithm for computing convolution with large filters. *CoRR abs/2102.13272* (2021).

[183] JIN, W., BARZILAY, R., AND JAAKKOLA, T. Junction tree variational autoencoder for molecular graph generation, 2018.

[184] JIN, X., YANG, Y., XU, N., YANG, J., JOJIC, N., FENG, J., AND YAN, S. WSNet: Compact and efficient networks through weight sampling. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 2352–2361.

[185] JUMPER, J., EVANS, R., PRITZEL, A., GREEN, T., FIGURNOV, M., RONNEBERGER, O., TUNYASUVUNAKOOL, K., BATES, R., ŽÍDEK, A., POTAPENKO, A., BRIDGLAND, A., MEYER, C., KOHL, S. A. A., BALLARD, A. J., COWIE, A., ROMERA-PAREDES, B., NIKOLOV, S., JAIN, R., ADLER, J., BACK, T., PETERSEN, S., REIMAN, D., CLANCY, E., ZIELINSKI, M., STEINEGGER, M., PACHOLSKA, M., BERGHAMMER, T., BODENSTEIN, S., SILVER, D., VINYALS, O., SENIOR, A. W., KAVUKCUOGLU, K., KOHLI, P., AND HASSABIS, D. Highly accurate protein structure prediction with alphafold. *Nature* (2021).

[186] JUNG, S., SON, C., LEE, S., SON, J., HAN, J.-J., KWAK, Y., HWANG, S. J., AND CHOI, C. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[187] KALCHBRENNER, N., ELSEN, E., SIMONYAN, K., NOURY, S., CASAGRANDE, N., LOCKHART, E., STIMBERG, F., VAN DEN OORD, A., DIELEMAN, S., AND KAVUKCUOGLU, K. Efficient neural audio synthesis. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 2410–2419.

[188] KARATSUBA, A. A. *The complexity of computations*. Nauka, Fizmatlit, Moscow, 1995, pp. 169–183.

[189] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1998), 359–392.

[190] KIM, S., KIM, M., SHIN, C., LEE, J., AND KIM, Y. Efficient implementation of ovsf code generator for umts systems. In *2009 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing* (Aug 2009), pp. 483–486.

[191] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)* (2015).

[192] KIPF, T. N., AND WELLING, M. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings* (2017), OpenReview.net.

[193] KLICPERA, J., BOJCHEVSKI, A., AND GÜNNEMANN, S. Combining neural networks with personalized pagerank for classification on graphs. In *International Conference on Learning Representations* (2019).

[194] KLYUCHNIKOV, N., TROFIMOV, I., ARTEMOVA, E., SALNIKOV, M., FEDOROV, M., AND BURNAEV, E. Nas-bench-nlp: Neural architecture search benchmark for natural language processing, 2020.

[195] KONECNÝ, J., MCMAHAN, H. B., YU, F. X., RICHTÁRIK, P., SURESH, A. T., AND BACON, D. Federated learning: Strategies for improving communication efficiency. *CoRR abs/1610.05492* (2016).

[196] KOURIS, A., VENIERIS, S. I., AND BOUGANIS, C. CascadeCNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), pp. 155–1557.

[197] KOURIS, A., VENIERIS, S. I., AND BOUGANIS, C. A Throughput-Latency Co-Optimised Cascade of Convolutional Neural Network Classifiers. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)* (2020), pp. 1656–1661.

[198] KRISHNAMOORTHI, R. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018.

[199] KRIZHEVSKY, A., ET AL. Learning multiple layers of features from tiny images. Tech. rep., Citeseer, 2009.

[200] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Red Hook, NY, USA, 2012), NIPS'12, Curran Associates Inc., p. 1097–1105.

[201] L. TOOM, A. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akademii Nauk SSSR 3* (01 1963).

[202] LAI, L., SUDA, N., AND CHANDRA, V. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus, 2018.

[203] LAI, L., SUDA, N., AND CHANDRA, V. Not all ops are created equal!, 2018.

[204] LAN, Z., CHEN, M., GOODMAN, S., GIMPEL, K., SHARMA, P., AND SORICUT, R. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations* (2020).

[205] LANE, N. D., BHATTACHARYA, S., GEORGIEV, P., FORLIVESI, C., JIAO, L., QENDRO, L., AND KAWSAR, F. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)* (2016), pp. 1–12.

[206] LANE, N. D., AND WARDEN, P. The deep (learning) transformation of mobile and embedded computing. *Computer 51*, 5 (May 2018), 12–16.

[207] LAVIN, A., AND GRAY, S. Fast algorithms for convolutional neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun 2016).

[208] LAVIN, P., YOUNG, J., VUDUC, R., RIEDY, J., VOSE, A., AND ERNST, D. *Evaluating Gather and Scatter Performance on CPUs and GPUs*. Association for Computing Machinery, New York, NY, USA, 2020, p. 209–222.

[209] LeCun, Y. The "MNIST" database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[210] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE* (1998), pp. 2278–2324.

[211] Lee, N., Ajanthan, T., Gould, S., and Torr, P. H. S. A signal propagation perspective for pruning neural networks at initialization. In *International Conference on Learning Representations* (2020).

[212] Lee, N., Ajanthan, T., and Torr, P. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations* (2019).

[213] Lee, R., Venieris, S. I., Dudziak, L., Bhattacharya, S., and Lane, N. D. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2019), MobiCom '19, Association for Computing Machinery.

[214] Lerer, A., Hu, H., Foerster, J., and Brown, N. Improving policies via search in cooperative partially observable games. *Proceedings of the AAAI Conference on Artificial Intelligence 34*, 05 (Apr. 2020), 7187–7194.

[215] Li, F., Zhang, B., and Liu, B. Ternary weight networks, 2016.

[216] Li, G., Liu, L., Wang, X., Ma, X., and Feng, X. Lance: efficient low-precision quantized winograd convolution for neural networks based on graphics processing units. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2020), pp. 3842–3846.

[217] Li, H., Bhargav, M., Whatmough, P. N., and Philip Wong, H. . On-Chip Memory Technology Design Space Explorations for Mobile Deep Neural Network Accelerators. In *2019 56th ACM/IEEE Design Automation Conference (DAC)* (June 2019), pp. 1–6.

[218] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets, 2017.

[219] Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-tzur, J., Hardt, M., Recht, B., and Talwalkar, A. A system for massively parallel hyperparameter tuning. In *Proceedings of Machine Learning and Systems 2020*. 2020, pp. 230–246.

[220] Li, T., Li, J., Liu, Z., and Zhang, C. Few sample knowledge distillation for efficient network compression, 2020.

[221] Li, T., Sahu, A. K., Zaheer, M., Sanjabi, M., Talwalkar, A., and Smith, V. Federated optimization in heterogeneous networks. In *Proceedings of Machine Learning and Systems* (2020), I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, pp. 429–450.

[222] Li, W., Chen, H., Huang, M., Chen, X., Xu, C., and Wang, Y. Winograd algorithm for addernet, 2021.

[223] Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118* (2018).

[224] Liberis, E., and Lane, N. D. Neural networks on microcontrollers: saving memory at inference via operator reordering, 2019.

[225] LIBERIS, E., ŁUKASZ DUDZIAK, AND LANE, N. D. μnas: Constrained neural architecture search for microcontrollers, 2020.

[226] LIN, T., KONG, L., STICH, S. U., AND JAGGI, M. Ensemble distillation for robust model fusion in federated learning. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 2351–2363.

[227] LIN, T.-Y., MAIRE, M., BELONGIE, S., BOURDEV, L., GIRSHICK, R., HAYS, J., PERONA, P., RAMANAN, D., ZITNICK, C. L., AND DOLLÁR, P. Microsoft coco: Common objects in context, 2015.

[228] LIN, X., ZHAO, C., AND PAN, W. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems* (2017), pp. 345–353.

[229] LIU, B., WANG, M., FOROOSH, H., TAPPEN, M., AND PENSKY, M. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015).

[230] LIU, H., SIMONYAN, K., AND YANG, Y. DARTS: Differentiable architecture search. In *International Conference on Learning Representations* (2019).

[231] LIU, S., LIN, Y., ZHOU, Z., NAN, K., LIU, H., AND DU, J. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2018), MobiSys '18, Association for Computing Machinery, p. 389–400.

[232] LIU, X., POOL, J., HAN, S., AND DALLY, W. J. Efficient sparse-winograd convolutional neural networks. In *International Conference on Learning Representations* (2018).

[233] LIU, Z., MU, H., ZHANG, X., GUO, Z., YANG, X., CHENG, K.-T., AND SUN, J. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE International Conference on Computer Vision* (2019), pp. 3296–3305.

[234] LIU, Z., WU, B., LUO, W., YANG, X., LIU, W., AND CHENG, K.-T. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 722–737.

[235] LIU, Z.-G., AND MATTINA, M. Efficient residue number system based winograd convolution. In *Computer Vision – ECCV 2020* (Cham, 2020), A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., Springer International Publishing, pp. 53–68.

[236] LOSHCHILOV, I., AND HUTTER, F. Sgdr: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations (ICLR) 2017 Conference Track* (Apr. 2017).

[237] LOUIZOS, C., ULLRICH, K., AND WELLING, M. Bayesian compression for deep learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017), NIPS'17, Curran Associates Inc., p. 3290–3300.

[238] LU, L., XIE, J., HUANG, R., ZHANG, J., LIN, W., AND LIANG, Y. An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2019), pp. 17–25.

[239] MA, Y., CAO, Y., VRUDHULA, S., AND SEO, J. Automatic Compilation of Diverse CNNs Onto High-Performance FPGA Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 39*, 2 (2020), 424–437.

[240] MA, Y., KIM, M., CAO, Y., VRUDHULA, S., AND SEO, J. End-to-End Scalable FPGA Accelerator for Deep Residual Networks. In *IEEE International Symposium on Circuits and Systems (ISCAS)* (2017), pp. 1–4.

[241] MAJI, P., MUNDY, A., DASIKA, G., BEU, J., MATTINA, M., AND MULLINS, R. Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus, 2019.

[242] MARTINEZ, B., YANG, J., BULAT, A., AND TZIMIROPOULOS, G. Training binary neural networks with real-to-binary convolutions. In *ICLR*. 2020.

[243] MATHIEU, M., HENAFF, M., AND LECUN, Y. Fast training of convolutional networks through ffts, 2013.

[244] MATHUR, A., BEUTEL, D. J., DE GUSMÃO, P. P. B., FERNANDEZ-MARQUES, J., TOPAL, T., QIU, X., PARCOLLET, T., GAO, Y., AND LANE, N. D. On-device federated learning with flower, 2021.

[245] MCINNES, L., HEALY, J., AND MELVILLE, J. Umap: Uniform manifold approximation and projection for dimension reduction, 2020.

[246] MCMAHAN, B., MOORE, E., RAMAGE, D., HAMPSON, S., AND Y ARCAS, B. A. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics* (2017), PMLR, pp. 1273–1282.

[247] MCMAHAN, H. B., MOORE, E., RAMAGE, D., AND Y ARCAS, B. A. Federated learning of deep networks using model averaging. *CoRR abs/1602.05629* (2016).

[248] MEHROTRA, A., RAMOS, A. G. C. P., BHATTACHARYA, S., DUDZIAK, Ł., VIPPERLA, R., CHAU, T., ABDELFATTAH, M. S., ISHTIAQ, S., AND LANE, N. D. {NAS}-bench-{asr}: Reproducible neural architecture search for speech recognition. In *International Conference on Learning Representations* (2021).

[249] MENG, L., AND BROTHERS, J. Efficient winograd convolution via integer arithmetic. *CoRR abs/1901.01965* (2019).

[250] MIRHOSEINI, A., GOLDIE, A., YAZGAN, M., JIANG, J., SONGHORI, E., WANG, S., LEE, Y.-J., JOHNSON, E., PATHAK, O., BAE, S., NAZI, A., PAK, J., TONG, A., SRINIVASA, K., HANG, W., TUNCER, E., BABU, A., LE, Q. V., LAUDON, J., HO, R., CARPENTER, R., AND DEAN, J. Chip placement with deep reinforcement learning, 2020.

[251] MIRZADEH, S. I., FARAJTABAR, M., LI, A., LEVINE, N., MATSUKAWA, A., AND GHASEMZADEH, H. Improved knowledge distillation via teacher assistant. *Proceedings of the AAAI Conference on Artificial Intelligence 34*, 04 (Apr. 2020), 5191–5198.

[252] MITTAL, D., BHARDWAJ, S., KHAPRA, M. M., AND RAVINDRAN, B. Recovering from random pruning: On the plasticity of deep convolutional neural networks, 2018.

[253] MOHAMED, A., OKHONKO, D., AND ZETTLEMOYER, L. Transformers with convolutional context for asr, 2020.

[254] MOHAN, P. V. *Residue Number Systems: Algorithms and Architectures*. Kluwer Academic Publishers, USA, 2002.

[255] MOHRI, M., PEREIRA, F., AND RILEY, M. Weighted finite-state transducers in speech recognition. *Computer Speech & Language 16*, 1 (2002), 69–88.

[256] MOLCHANOV, D., ASHUKHA, A., AND VETROV, D. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning* (06–11 Aug 2017), D. Precup and Y. W. Teh, Eds., vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 2498–2507.

[257] MOLCHANOV, P., MALLYA, A., TYREE, S., FROSIO, I., AND KAUTZ, J. Importance Estimation for Neural Network Pruning. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[258] MONTGOMERIE-CORCORAN, A., AND SAVVAS-BOUGANIS, C. DEF: Differential Encoding of Featuremaps for Low Power Convolutional Neural Network Accelerators. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC* (2021).

[259] MONTI, F., BOSCAINI, D., MASCI, J., RODOLA, E., SVOBODA, J., AND BRONSTEIN, M. M. Geometric deep learning on graphs and manifolds using mixture model cnns. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Los Alamitos, CA, USA, jul 2017), IEEE Computer Society, pp. 5425–5434.

[260] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., ELIBOL, M., YANG, Z., PAUL, W., JORDAN, M. I., AND STOICA, I. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 561–577.

[261] MOZER, M. C., AND SMOLENSKY, P. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Proceedings of the 1st International Conference on Neural Information Processing Systems* (Cambridge, MA, USA, 1988), NIPS'88, MIT Press, p. 107–115.

[262] MUKKARA, A., BECKMANN, N., ABEYDEERA, M., MA, X., AND SANCHEZ, D. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (2018), MICRO-51, IEEE Press, p. 1–14.

[263] NAGEL, M., FOURNARAKIS, M., AMJAD, R. A., BONDARENKO, Y., VAN BAALEN, M., AND BLANKEVOORT, T. A white paper on neural network quantization, 2021.

[264] NAHSHAN, Y., CHMIEL, B., BASKIN, C., ZHELTONOZHSKII, E., BANNER, R., BRONSTEIN, A. M., AND MENDELSON, A. Loss aware post-training quantization, 2020.

[265] NAIR, A., ROY, A., AND MEINKE, K. Funcgnn: A graph neural network approach to program similarity. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (New York, NY, USA, 2020), ESEM '20, Association for Computing Machinery.

[266] NIU, Y., KANNAN, R., SRIVASTAVA, A., AND PRASANNA, V. Reuse Kernels or Activations? A Flexible Dataflow for Low-Latency Spectral CNN Acceleration. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (2020), p. 266–276.

[267] OZEROV, A., AND DUONG, N. Inplace knowledge distillation with teacher assistant for improved training of flexible deep neural networks, 2021.

[268] PAN, V. Y. How bad are vandermonde matrices? *SIAM Journal on Matrix Analysis and Applications 37*, 2 (2016), 676–694.

[269] PANAYOTOV, V., CHEN, G., POVEY, D., AND KHUDANPUR, S. Librispeech: an asr corpus based on public domain audio books. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on* (2015), IEEE, pp. 5206–5210.

[270] PARASHAR, A., RHU, M., MUKKARA, A., PUGLIELLI, A., VENKATESAN, R., KHAILANY, B., EMER, J., KECKLER, S. W., AND DALLY, W. J. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 27–40.

[271] PARCOLLET, T., QIU, X., AND LANE, N. D. FusionRNN: Shared Neural Parameters for Multi-Channel Distant Speech Recognition. In *Proc. Interspeech 2020* (2020), pp. 1678–1682.

[272] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop* (2017).

[273] PFAFF, T., FORTUNATO, M., SANCHEZ-GONZALEZ, A., AND BATTAGLIA, P. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations* (2021).

[274] PRATO, G., CHARLAIX, E., AND REZAGHOLIZADEH, M. Fully quantized transformer for machine translation, 2019.

[275] PUROHIT, G., CHAUBEY, V. K., RAJU, K. S., AND REDDY, P. V. Fpga based implementation and testing of ovsf code. In *2013 International Conference on Advanced Electronic Systems (ICAES)* (Sept 2013), pp. 88–92.

[276] QIN, H., GONG, R., LIU, X., BAI, X., SONG, J., AND SEBE, N. Binary neural networks: A survey. *Pattern Recognition 105* (2020), 107281.

[277] QIU, Q., CHENG, X., CALDERBANK, R., AND SAPIRO, G. DCFNet: Deep neural network with decomposed convolutional filters. *International Conference on Machine Learning* (2018).

[278] QIU, X., PARCOLLET, T., FERNANDEZ-MARQUES, J., DE GUSMAO, P. P. B., BEUTEL, D. J., TOPAL, T., MATHUR, A., AND LANE, N. D. A first look into the carbon footprint of federated learning, 2021.

[279] RAJPURKAR, P., JIA, R., AND LIANG, P. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Melbourne, Australia, July 2018), Association for Computational Linguistics, pp. 784–789.

[280] RAMACHANDRAN, P., ZOPH, B., AND LE, Q. V. Searching for activation functions, 2018.

[281] RASTEGARI, M., ORDONEZ, V., REDMON, J., AND FARHADI, A. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR abs/1603.05279* (2016).

[282] REAGAN, B., GUPTA, U., ADOLF, B., MITZENMACHER, M., RUSH, A., WEI, G.-Y., AND BROOKS, D. Weightless: Lossy weight encoding for deep neural network compression. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 4324–4333.

[283] REAL, E., AGGARWAL, A., HUANG, Y., AND LE, Q. V. Regularized evolution for image classifier architecture search. *Proceedings of the AAAI Conference on Artificial Intelligence 33*, 01 (Jul. 2019), 4780–4789.

[284] REDDI, V. J., CHENG, C., KANTER, D., MATTSON, P., SCHMUELLING, G., WU, C.-J., ANDERSON, B., BREUGHE, M., CHARLEBOIS, M., CHOU, W., CHUKKA, R., COLEMAN, C., DAVIS, S., DENG, P., DIAMOS, G., DUKE, J., FICK, D., GARDNER, J. S., HUBARA, I., IDGUNJI, S., JABLIN, T. B., JIAO, J., JOHN, T. S., KANWAR, P., LEE, D., LIAO, J., LOKHMOTOV, A., MASSA, F., MENG, P., MICIKEVICIUS, P., OSBORNE, C., PEKHIMENKO, G., RAJAN, A. T. R., SEQUEIRA,

D., SIRASAO, A., SUN, F., TANG, H., THOMSON, M., WEI, F., WU, E., XU, L., YAMADA, K., YU, B., YUAN, G., ZHONG, A., ZHANG, P., AND ZHOU, Y. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), pp. 446–459.

[285] REN, M., POKROVSKY, A., YANG, B., AND URTASUN, R. Sbnet: Sparse blocks network for fast inference, 2018.

[286] RINTAKOSKI, T., KUULUSA, M., AND NURMI, J. Hardware unit for ovsf/walsh/hadamard code generation [3g mobile communication applications]. In *2004 International Symposium on System-on-Chip, 2004. Proceedings.* (Nov 2004), pp. 143–145.

[287] ROBERT DAVID, JARED DUKE, A. J. V. J. R. N. J. J. L. N. K. I. N. M. N. S. R. R. R. T. W., AND WARDEN, P. Tensorflow lite micro: Embedded machine learning for tinyml systems, 2021.

[288] ROCHETEAU, E., TONG, C., VELIČKOVIĆ, P., LANE, N., AND LIÒ, P. Predicting patient outcomes with graph representation learning, 2021.

[289] RONG, Y., HUANG, W., XU, T., AND HUANG, J. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations* (2020).

[290] RUSCI, M., ROSSI, D., FARELLA, E., AND BENINI, L. A sub-mw iot-endnode for always-on visual monitoring and smart triggering. *IEEE Internet of Things Journal 4*, 5 (Oct 2017), 1284–1295.

[291] S. GRAY, A. R., AND KINGMA, D. P. Block-sparse gpu kernels, 2017.

[292] SAHU, A. K., LI, T., SANJABI, M., ZAHEER, M., TALWALKAR, A., AND SMITH, V. On the convergence of federated optimization in heterogeneous networks. *CoRR abs/1812.06127* (2018).

[293] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Jun 2018).

[294] SARLIN, P.-E., DETONE, D., MALISIEWICZ, T., AND RABINOVICH, A. Superglue: Learning feature matching with graph neural networks. *arXiv preprint arXiv:1911.11763* (2019).

[295] SEHWAG, V., WANG, S., MITTAL, P., AND JANA, S. Hydra: Pruning adversarially robust neural networks. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 19655–19666.

[296] SENIOR, A. W., EVANS, R., JUMPER, J., KIRKPATRICK, J., SIFRE, L., GREEN, T., QIN, C., ŽÍDEK, A., NELSON, A. W. R., BRIDGLAND, A., PENEDONES, H., PETERSEN, S., SIMONYAN, K., CROSSAN, S., KOHLI, P., JONES, D. T., SILVER, D., KAVUKCUOGLU, K., AND HASSABIS, D. Improved protein structure prediction using potentials from deep learning. *Nature 577*, 7792 (2020), 706–710.

[297] SHEN, Y., FERDMAN, M., AND MILDER, P. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2017), pp. 93–100.

[298] SHEN, Y., FERDMAN, M., AND MILDER, P. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)* (2017).

[299] SHENG, T., FENG, C., ZHUO, S., ZHANG, X., SHEN, L., AND ALEKSIC, M. A quantization-friendly separable convolution for mobilenets. *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)* (Mar 2018).

[300] SHI, R., DING, Y., WEI, X., LI, H., LIU, H., SO, H. K. . H., AND DING, C. FTDL: A Tailored FPGA-Overlay for Deep Learning with High Scalability. In *57th ACM/IEEE Design Automation Conference (DAC)* (2020), pp. 1–6.

[301] SHOEYBI, M., PATWARY, M., PURI, R., LEGRESLEY, P., CASPER, J., AND CATANZARO, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[302] SHUMAN, D. I., NARANG, S. K., FROSSARD, P., ORTEGA, A., AND VANDERGHEYNST, P. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine 30*, 3 (May 2013), 83–98.

[303] SIEKKINEN, M., HIIENKARI, M., NURMINEN, J. K., AND NIEMINEN, J. How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4. In *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)* (April 2012), pp. 232–237.

[304] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., DIELEMAN, S., GREWE, D., NHAM, J., KALCHBRENNER, N., SUTSKEVER, I., LILLICRAP, T., LEACH, M., KAVUKCUOGLU, K., GRAEPEL, T., AND HASSABIS, D. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[305] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations* (2015).

[306] SOLEIMAN, A., AND VARSHNEY, A. Towards backscatter-enabled networked utensils (poster). In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2019), MobiSys '19, Association for Computing Machinery, p. 537–538.

[307] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[308] SRIVASTAVA, N., JIN, H., SMITH, S., RONG, H., ALBONESI, D., AND ZHANG, Z. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), pp. 689–702.

[309] STOCK, P., FAN, A., GRAHAM, B., GRAVE, E., GRIBONVAL, R., JEGOU, H., AND JOULIN, A. Training with quantization noise for extreme model compression. In *International Conference on Learning Representations* (2021).

[310] STRASSEN, V. Gaussian elimination is not optimal. *Numer. Math. 13*, 4 (Aug. 1969), 354–356.

[311] SUAREZ, J. Language modeling with recurrent highway hypernetworks. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[312] SUAU, X., ZAPPELLA, L., AND APOSTOLOFF, N. Filter distillation for network compression, 2019.

[313] SUN, C., SHRIVASTAVA, A., SINGH, S., AND GUPTA, A. Revisiting unreasonable effectiveness of data in deep learning era. In *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), pp. 843–852.

[314] SUTSKEVER, I., MARTENS, J., DAHL, G., AND HINTON, G. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning* (Atlanta, Georgia, USA, 17–19 Jun 2013), S. Dasgupta and D. McAllester, Eds., vol. 28 of *Proceedings of Machine Learning Research*, PMLR, pp. 1139–1147.

[315] SZE, V., CHEN, Y.-H., YANG, T.-J., AND EMER, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE 105*, 12 (Dec 2017), 2295–2329.

[316] TAILOR, S. A., FERNANDEZ-MARQUES, J., AND LANE, N. D. Degree-quant: Quantization-aware training for graph neural networks. In *International Conference on Learning Representations* (2021).

[317] TAILOR, S. A., OPOLKA, F. L., LIÒ, P., AND LANE, N. D. Adaptive filters and aggregator fusion for efficient graph convolutions, 2021. GNNSys Workshop, MLSys '21 and HAET Workshop, ICLR '21.

[318] TAN, M., CHEN, B., PANG, R., VASUDEVAN, V., AND LE, Q. V. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR* (2019).

[319] TAN, M., AND LE, Q. V. Efficientnet: Rethinking model scaling for convolutional neural networks, 2019.

[320] THEKUMPARAMPIL, K. K., OH, S., WANG, C., AND LI, L.-J. Attention-based graph neural network for semi-supervised learning, 2018.

[321] TONG, T., LI, G., LIU, X., AND GAO, Q. Image super-resolution using dense skip connections. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (Oct 2017).

[322] TOUVRON, H., CORD, M., DOUZE, M., MASSA, F., SABLAYROLLES, A., AND JÉGOU, H. Training data-efficient image transformers distillation through attention, 2021.

[323] TSCHANNEN, M., KHANNA, A., AND ANANDKUMAR, A. StrassenNets: Deep learning with a multiplication budget. In *Proceedings of the 35th International Conference on Machine Learning* (Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 4985–4994.

[324] TSENG, V. W.-S., BHATTACHARYA, S., MARQUÉS, J. F., ALIZADEH, M., TONG, C., AND LANE, N. D. Deterministic Binary Filters for Convolutional Neural Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI)* (2018), pp. 2739–2747.

[325] VAN AMERSFOORT, J., ALIZADEH, M., FARQUHAR, S., LANE, N., AND GAL, Y. Single shot structured pruning before training, 2020.

[326] VAN DEN BERG, R., KIPF, T. N., AND WELLING, M. Graph convolutional matrix completion, 2017.

[327] VARSHNEY, A., SOLEIMAN, A., AND VOIGT, T. Tunnelscatter: Low power communication for sensor tags using tunnel diodes. In *The 25th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2019), MobiCom '19, Association for Computing Machinery.

[328] VASUDEVAN, A., ANDERSON, A., AND GREGG, D. Parallel multi channel convolution using general matrix multiplication. *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (Jul 2017).

[329] VELICKOVIC, P., CUCURULL, G., CASANOVA, A., ROMERO, A., LIÒ, P., AND BENGIO, Y. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings* (2018), OpenReview.net.

[330] VELIČKOVIĆ, P., BUESING, L., OVERLAN, M., PASCANU, R., VINYALS, O., AND BLUNDELL, C. Pointer graph networks. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 2232–2244.

[331] VELIČKOVIĆ, P., FEDUS, W., HAMILTON, W. L., LIÒ, P., BENGIO, Y., AND HJELM, R. D. Deep graph infomax. In *International Conference on Learning Representations* (2019).

[332] VENIERIS, S. I., AND BOUGANIS, C. Latency-Driven Design for FPGA-based Convolutional Neural Networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (2017), pp. 1–8.

[333] VENIERIS, S. I., FERNANDEZ-MARQUES, J., AND LANE, N. D. unzipfpga: Enhancing fpga-based cnn engines with on-the-fly weights generation. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Los Alamitos, CA, USA, may 2021), IEEE Computer Society, pp. 165–175.

[334] VERELST, T., AND TUYTELAARS, T. Segblocks: Block-based dynamic resolution networks for real-time segmentation, 2020.

[335] VINCENT, K., STEPHANO, K., FRUMKIN, M., GINSBURG, B., AND J., D. On improving the numerical stability of winograd convolutions. In *International Conference on Learning Representations (Workshop track)* (2017).

[336] VINYALS, O., BABUSCHKIN, I., CZARNECKI, W. M., MATHIEU, M., DUDZIK, A., CHUNG, J., CHOI, D. H., POWELL, R., EWALDS, T., GEORGIEV, P., OH, J., HORGAN, D., KROISS, M., DANIHELKA, I., HUANG, A., SIFRE, L., CAI, T., AGAPIOU, J. P., JADERBERG, M., VEZHNEVETS, A. S., LEBLOND, R., POHLEN, T., DALIBARD, V., BUDDEN, D., SULSKY, Y., MOLLOY, J., PAINE, T. L., GULCEHRE, C., WANG, Z., PFAFF, T., WU, Y., RING, R., YOGATAMA, D., WÜNSCH, D., MCKINNEY, K., SMITH, O., SCHAUL, T., LILLICRAP, T., KAVUKCUOGLU, K., HASSABIS, D., APPS, C., AND SILVER, D. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature 575*, 7782 (2019), 350–354.

[337] VIOLA, P., AND JONES, M. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001* (2001), vol. 1, pp. I–I.

[338] VOGEL, S., LIANG, M., GUNTORO, A., STECHELE, W., AND ASCHEID, G. Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base. In *Proceedings of the International Conference on Computer-Aided Design* (New York, NY, USA, 2018), ICCAD '18, Association for Computing Machinery.

[339] VON OSWALD, J., HENNING, C., SACRAMENTO, J., AND GREWE, B. F. Continual learning with hypernetworks. In *International Conference on Learning Representations* (2020).

[340] VÉNIAT, T., AND DENOYER, L. Learning time/memory-efficient deep architectures with budgeted super networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 3492–3500.

[341] WAN, D., SHEN, F., LIU, L., SHAO, L., QIN, J., AND SHEN, H. T. Tbn: Convolutional neural network with ternary inputs and binary weights. In *ECCV* (2018).

[342] WANG, A., SINGH, A., MICHAEL, J., HILL, F., LEVY, O., AND BOWMAN, S. R. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations* (2019).

[343] WANG, C., ZHANG, G., AND GROSSE, R. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations* (2020).

[344] WANG, J., BAO, W., SUN, L., ZHU, X., CAO, B., AND YU, P. S. Private model compression via knowledge distillation. *Proceedings of the AAAI Conference on Artificial Intelligence 33*, 01 (Jul. 2019), 1190–1197.

[345] WANG, J., LIU, W., KUMAR, S., AND CHANG, S.-F. Learning to hash for indexing big data—a survey. *Proceedings of the IEEE 104*, 1 (2016), 34–57.

[346] WANG, J., WANG, Y., YANG, Z., YANG, L., AND GUO, Y. Bi-gcn: Binary graph convolutional network, 2021.

[347] WANG, K., LIU, Z., LIN, Y., LIN, J., AND HAN, S. Haq: Hardware-aware automated quantization with mixed precision, 2018.

[348] WANG, K., LIU, Z., LIN, Y., LIN, J., AND HAN, S. Haq: Hardware-aware automated quantization with mixed precision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[349] WANG, M., RASOULINEZHAD, S., LEONG, P. H. W., AND SO, H. K. H. Niti: Training integer neural networks using integer-only arithmetic, 2020.

[350] WANG, P., CHEN, Q., HE, X., AND CHENG, J. Towards accurate post-training network quantization via bit-split and stitching. In *Proceedings of the 37th International Conference on Machine Learning* (13–18 Jul 2020), H. D. III and A. Singh, Eds., vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 9847–9856.

[351] WANG, Y., AND KARALETSOS, T. Stochastic aggregation in graph neural networks, 2021.

[352] WANG, Y., SUN, Y., LIU, Z., SARMA, S. E., BRONSTEIN, M. M., AND SOLOMON, J. M. Dynamic graph cnn for learning on point clouds. *ACM Trans. Graph. 38*, 5 (Oct. 2019).

[353] WANG, Y., XU, C., XU, C., AND TAO, D. Beyond Filters: Compact Feature Map for Portable Deep Model. In *Proceedings of the 34th International Conference on Machine Learning (ICML)* (2017), pp. 3703–3711.

[354] WANG, Y., XU, C., YOU, S., TAO, D., AND XU, C. Cnnpack: Packing convolutional neural networks in the frequency domain. In *Advances in Neural Information Processing Systems* (2016), D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29, Curran Associates, Inc.

[355] WANG, Z. Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2020), PACT '20, Association for Computing Machinery, p. 31–42.

[356] WANG, Z. Sparsednn: Fast sparse deep learning inference on cpus, 2021.

[357] WANG, Z., ROSA, S., YANG, B., WANG, S., TRIGONI, N., AND MARKHAM, A. 3d-physnet: Learning the intuitive physics of non-rigid object deformations. In *27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence IJCAI-ECAI* (2018).

[358] WARDEN, P. Speech commands: A public dataset for single-word speech recognition.

[359] WARDEN, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *ArXiv e-prints* (Apr. 2018).

[360] WASKOM, M. L. seaborn: statistical data visualization. *Journal of Open Source Software 6*, 60 (2021), 3021.

[361] WATTERS, N., TACCHETTI, A., WEBER, T., PASCANU, R., BATTAGLIA, P., AND ZORAN, D. Visual interaction networks: Learning a physics simulator from video. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017), NIPS'17, Curran Associates Inc., p. 4542–4550.

[362] WEI, H., YU, J. X., LU, C., AND LIN, X. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1813–1828.

[363] WEISFEILER, B., AND LEHMAN, A. A reduction of a graph to a canonical form and an algebra arising during this reduction, 1968.

[364] WEN, N., GUO, R., HE, B., FAN, Y., AND MA, D. Block-sparse cnn: towards a fast and memory-efficient framework for convolutional neural networks. *Applied Intelligence 51* (2020), 441–452.

[365] WHATMOUGH, P. N., LEE, S. K., BROOKS, D., AND WEI, G. DNN Engine: A 28-nm Timing-Error Tolerant Sparse Deep Neural Network Processor for IoT Applications. *IEEE Journal of Solid-State Circuits 53*, 9 (Sep. 2018), 2722–2731.

[366] WHATMOUGH, P. N., ZHOU, C., HANSEN, P., VENKATARAMANAIAH, S. K., SEO, J.-S., AND MATTINA, M. FixyNN: Efficient Hardware for Mobile Computer Vision via Transfer Learning, 2019.

[367] WINOGRAD, S. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980.

[368] WINOGRAD, S. Signal processing and complexity of computation. In *ICASSP '80. IEEE International Conference on Acoustics, Speech, and Signal Processing* (April 1980), vol. 5, pp. 94–101.

[369] WOO, J., OK, J., AND YI, Y. Iterative learning of graph connectivity from partially-observed cascade samples. In *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing* (New York, NY, USA, 2020), Mobihoc '20, Association for Computing Machinery, p. 141–150.

[370] WU, B., DAI, X., ZHANG, P., WANG, Y., SUN, F., WU, Y., TIAN, Y., VAJDA, P., JIA, Y., AND KEUTZER, K. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[371] WU, F., SOUZA, A., ZHANG, T., FIFTY, C., YU, T., AND WEINBERGER, K. Simplifying graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning* (09–15 Jun 2019), K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 6861–6871.

[372] WU, H., JUDD, P., ZHANG, X., ISAEV, M., AND MICIKEVICIUS, P. Integer quantization for deep learning inference: Principles and empirical evaluation, 2020.

[373] WU, J., WANG, Y., WU, Z., WANG, Z., VEERARAGHAVAN, A., AND LIN, Y. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 5363–5372.

[374] WU, Z., RAMSUNDAR, B., FEINBERG, E. N., GOMES, J., GENIESSE, C., PAPPU, A. S., LESWING, K., AND PANDE, V. Moleculenet: A benchmark for molecular machine learning, 2017.

[375] WU, Z., SONG, S., KHOSLA, A., YU, F., ZHANG, L., TANG, X., AND XIAO, J. 3d shapenets: A deep representation for volumetric shapes. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 1912–1920.

[376] XIANG, X., QIAN, Y., AND YU, K. Binary deep neural networks for speech recognition. In *Proc. Interspeech 2017* (2017), pp. 533–537.

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

[378] Xie, S., Kirillov, A., Girshick, R., and He, K. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (October 2019).

[379] Xing, Y., Liang, S., Sui, L., Jia, X., Qiu, J., Liu, X., Wang, Y., Shan, Y., and Wang, Y. DNNVM: End-to-End Compiler Leveraging Heterogeneous Optimizations on FPGA-based CNN Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[380] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., Yu, D., and Zweig, G. Achieving human parity in conversational speech recognition, 2017.

[381] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (2019), OpenReview.net.

[382] Xu, Y., Li, Y., Zhang, S., Wen, W., Wang, B., Qi, Y., Chen, Y., Lin, W., and Xiong, H. Trained rank pruning for efficient deep neural networks, 2020.

[383] Xue, J., Li, J., and Gong, Y. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech* (January 2013).

[384] Yalniz, I. Z., Jégou, H., Chen, K., Paluri, M., and Mahajan, D. Billion-scale semi-supervised learning for image classification, 2019.

[385] Yan, B., Wang, C., Guo, G., and Lou, Y. Tinygnn: Learning efficient graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2020), KDD '20, Association for Computing Machinery, p. 1848–1856.

[386] Yan, M., Chen, Z., Deng, L., Ye, X., Zhang, Z., Fan, D., and Xie, Y. Characterizing and understanding gcns on gpu, 2020.

[387] Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., Zhang, Z., Fan, D., and Xie, Y. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), pp. 15–29.

[388] Yang, J., Lu, J., Lee, S., Batra, D., and Parikh, D. Graph r-cnn for scene graph generation. In *Proceedings of the European Conference on Computer Vision (ECCV)* (2018), pp. 670–685.

[389] Yang, J., Shen, X., Xing, J., Tian, X., Li, H., Deng, B., Huang, J., and Hua, X.-s. Quantization networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[390] Yang, J., Zou, H., Cao, S., Chen, Z., and Xie, L. Mobileda: Toward edge-domain adaptation. *IEEE Internet of Things Journal 7*, 8 (2020), 6909–6918.

[391] Yang, T., Chen, Y., and Sze, V. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Los Alamitos, CA, USA, jul 2017), IEEE Computer Society, pp. 6071–6079.

[392] Yang, Y., Qiu, J., Song, M., Tao, D., and Wang, X. Distilling knowledge from graph convolutional networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2020).

[393] YANG, Y., YU, J., JOJIC, N., HUAN, J., AND HUANG, T. S. FSNet: Compression of Deep Convolutional Neural Networks by Filter Summary. In *International Conference on Learning Representations (ICLR)* (2020).

[394] YANG, Z., DAI, Z., YANG, Y., CARBONELL, J., SALAKHUTDINOV, R. R., AND LE, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.

[395] YANJUN MA, DIANHAI YU, T. W., AND WANG, H. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Computing 1*, 1 (2019), 105.

[396] YIN, P., NEUBIG, G., ALLAMANIS, M., BROCKSCHMIDT, M., AND GAUNT, A. L. Learning to represent edits. In *International Conference on Learning Representations* (2019).

[397] YING, C., KLEIN, A., CHRISTIANSEN, E., REAL, E., MURPHY, K., AND HUTTER, F. NAS-bench-101: Towards reproducible neural architecture search. In *Proceedings of the 36th International Conference on Machine Learning* (Long Beach, California, USA, 09–15 Jun 2019), K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 7105–7114.

[398] YING, R., HE, R., CHEN, K., EKSOMBATCHAI, P., HAMILTON, W. L., AND LESKOVEC, J. Graph convolutional neural networks for web-scale recommender systems. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (Jul 2018).

[399] YU, J., LUKEFAHR, A., PALFRAMAN, D., DASIKA, G., DAS, R., AND MAHLKE, S. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), p. 548–560.

[400] YU, R., LI, A., CHEN, C., LAI, J., MORARIU, V. I., HAN, X., GAO, M., LIN, C., AND DAVIS, L. S. Nisp: Pruning networks using neuron importance score propagation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (Los Alamitos, CA, USA, jun 2018), IEEE Computer Society, pp. 9194–9203.

[401] ZACHARIADIS, O., SATPUTE, N., GÓMEZ-LUNA, J., AND OLIVARES, J. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers Electrical Engineering 88* (Dec 2020), 106848.

[402] ZAFRIR, O., BOUDOUKH, G., IZSAK, P., AND WASSERBLAT, M. Q8bert: Quantized 8bit bert, 2019.

[403] ZELLERS, R., YATSKAR, M., THOMSON, S., AND CHOI, Y. Neural motifs: Scene graph parsing with global context. In *Conference on Computer Vision and Pattern Recognition* (2018).

[404] ZENG, H., AND PRASANNA, V. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2020), FPGA '20, Association for Computing Machinery, p. 255–265.

[405] ZENG, H., ZHOU, H., SRIVASTAVA, A., KANNAN, R., AND PRASANNA, V. Graphsaint: Graph sampling based inductive learning method, 2020.

[406] ZHANG, B., ZENG, H., AND PRASANNA, V. Hardware acceleration of large scale gcn inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2020), pp. 61–68.

[407] ZHANG, C., LI, P., SUN, G., GUAN, Y., XIAO, B., AND CONG, J. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* (2015), p. 161–170.

[408] ZHANG, C., SUN, G., FANG, Z., ZHOU, P., PAN, P., AND CONG, J. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 38*, 11 (2019), 2072–2085.

[409] ZHANG, J., SHIH, K. J., ELGAMMAL, A., TAO, A., AND CATANZARO, B. Graphical contrastive losses for scene graph parsing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[410] ZHANG, S., DU, Z., ZHANG, L., LAN, H., LIU, S., LI, L., GUO, Q., CHEN, T., AND CHEN, Y. Cambricon-X: An Accelerator for Sparse Neural Networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), pp. 1–12.

[411] ZHANG, X., ZHOU, X., LIN, M., AND SUN, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Jun 2018).

[412] ZHANG, X., ZOU, J., MING, X., HE, K., AND SUN, J. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015).

[413] ZHANG, Y., LI, K., LI, K., WANG, L., ZHONG, B., AND FU, Y. Image super-resolution using very deep residual channel attention networks. *CoRR abs/1807.02758* (2018).

[414] ZHANG, Y., SUDA, N., LAI, L., AND CHANDRA, V. Hello edge: Keyword spotting on microcontrollers, 2018.

[415] ZHANG, Y., TIAN, Y., KONG, Y., ZHONG, B., AND FU, Y. Residual dense network for image super-resolution. In *CVPR* (2018).

[416] ZHANG, Z., YANG, J., AND ZHAO, H. Retrospective reader for machine reading comprehension, 2020.

[417] ZHOU, X., DU, Z., GUO, Q., LIU, S., LIU, C., WANG, C., ZHOU, X., LI, L., CHEN, T., AND CHEN, Y. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018), pp. 15–28.

[418] ZHU, M. H., AND GUPTA, S. To prune, or not to prune: Exploring the efficacy of pruning for model compression, 2018.

[419] ZHU, R., ZHAO, K., YANG, H., LIN, W., ZHOU, C., AI, B., LI, Y., AND ZHOU, J. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow. 12*, 12 (Aug. 2019), 2094–2105.

[420] ZOPH, B., AND LE, Q. V. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations* (2017).

[421] ZOPH, B., VASUDEVAN, V., SHLENS, J., AND LE, Q. V. Learning transferable architectures for scalable image recognition. *CoRR abs/1707.07012* (2017).

Appendix A

A.1 Collaboration Acknowledgments

In this section I list the works performed during my DPhil studies and provide details on my concrete contributions:

[324] Vincent Tseng, S. Bhattacharya, **Javier Fernandez-Marques**, Milad Alizadeh, Catherine Tong and Nicholas D. Lane. *Deterministic Binary Filters for Convolutional Neural Networks.* In *Proceedings of the Twnety-Seventh International Joint Conference on Artificial Intelligence* (IJCAI), 2018.
I also presented this work in MobiUK 2018 and Arm Research Summit 2018.
After two unsuccessful publication attempts at other international conferences, I lead the re-design of the paper and its core contributions. I shifted its framing from being a network architecture design into a general formulation that could be adapted to all types of networks relying on convolutional layers. I also addressed some of the early limitations identified when using the proposed deterministic codes, paving the way for further research that later materialised into [102] and [333].

[103] **Javier Fernandez-Marques** , Vincent W.-S. Tseng, Sourav Bhattacharya and Nicholas D. Lane. *BinaryCmd: Keyword Spotting with deterministic binary basis*. In *Conference on Machine Learning and Systems* (MLSys), 2018.
A follow-up version of this work was presented in EMDL-MobiSys 2018. In this work I study the embedding of on-the-fly layers for key word spotting applications running on Cortex-M microcontrollers, a very compute and memory constrained setting. I further study some of the limitations of using OVSF codes as a base for on-the-fly models and propose an effective way to ease these.

[333] Stylianos I. Venieris*, **Javier Fernandez-Marques*** and Nicholas D. Lane. *unzipFPGA: Enhancing FPGA-based CNN Engines with On-the-fly Weights Generation.* In *IEEE International Symposium on Field-Programmable Custom Computing Machines* (FCCM), 2021.

In this work we present a hardware weights generation following the on-the-fly formulation I proposed in [324] and [102]. My contributions in this work are: design a better training strategy for models using on-the-fly layers that reach equal performance to ImageNet-level baselines; propose two solutions to the power-of-two limitation of OVSF deterministic codes; implement and run all experiments assessing the performance in terms of accuracy of the various model architectures and baselines considered; regarding the FPGA-based implementation, while the majority of the design and evaluation was done by Stylianos, I contributed in the framing of the comparisons with the reference implementations and pruned baselines as wells as the discussion on multi-tenant FPGA designs. I planned having a bigger involvement on the hardware dimension of this project but Covid-19 got on the way.

[104] **Javier Fernandez-Marques**, Paul N. Whatmough, Andrew Mundy and Matthew Mattina. *Searching for Winograd-aware Quantized Networks.* In *Proceedings of Machine Learning and Systems* (MLSys), 2020.
In this work I propose a relaxation on the formulation of Winograd convolutions that allowed training quantized models that do not suffer from numerical degradation when using this algorithm. I lead this project, implemented code and ran an extensive set of experiments. To perform the latency benchmark, Andrew Mundy helped with relevant code making use of Arm Compute Library. Most of the contents presented in chapter 4 are the result of my 4-month summer internship at Arm. I carried out some of the experiments and associated discussion after my internship.

[316] Shyam A. Tailor*, **Javier Fernandez-Marques*** and Nicholas D. Lane. *Degree-Quant: Quantization-Aware Training for Graph Neural Networks.* In *International Conference on Learning Representations* (ICLR), 2021.
Also in *Graph Representation Learning and Beyond Workshop* (ICMLw), 2020.
In this work we propose the first general formulation for quantization-aware training on graphs. My contributions are: propose and run the study assessing the performance of GNNs with off-the-shelf quantization implementations; implement and run all quantization baselines, leading to a total of 200 individual experiments over 6 datasets; streamline our framework relying on Ray Tune and PyG for our multi-node SLURM + Docker setup; and, collaborate defining the unique features of the proposed `Degree-Quant` framework, ablation studies and, overall framing of the paper. The content I present in Chapter 5 expands upon what was first introduced in [316] providing further context, comparisons and, new experiments.

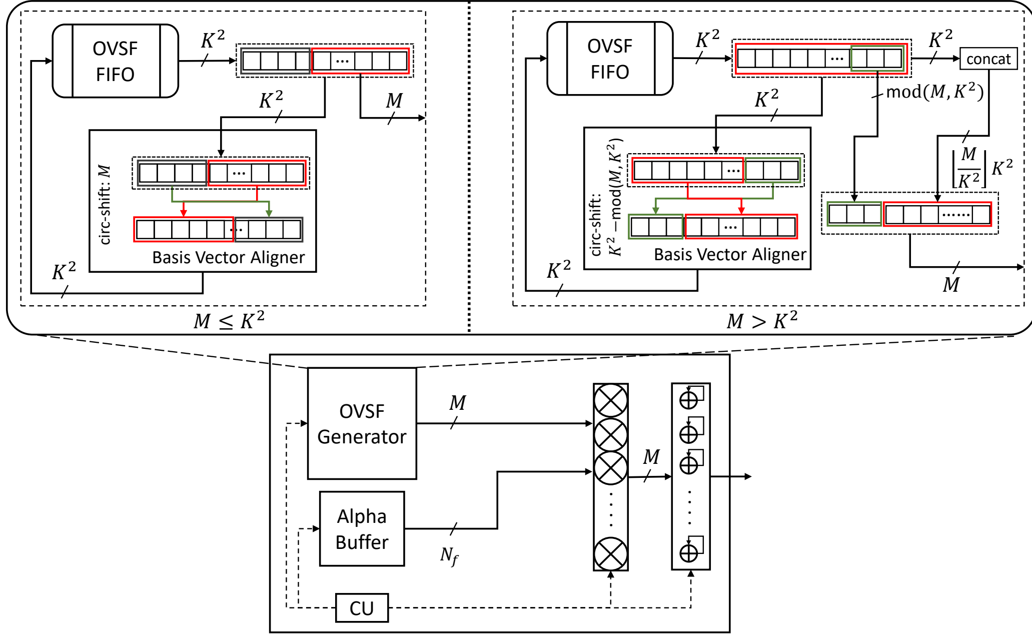


Figure A.1: Microarchitecture of the `CNN-WGen` module with a detailed representation of the OVSF code generator. In Section 3.3.3 all the elements in `CNN-WGen` are described.

For the other works not included as part of this thesis, my contributions were as follows: in [11], I helped Milad carry out some of the experiments during the last couple of weeks before the ICLR deadline and for rebuttal; in [110, 278] I played a support role helping with experiments and by implementing an efficient *virtualization* of federated learning clients tool, which allowed running thousands of clients regardless of the compute resources available on the servers we had at our disposal. I presented this tool under the name of *Virtual Client Manager* in the Flower Summit 2021 [9] and, is currently being integrated into the main Flower repository. Finally, in [244] I helped with deploying Flower clients on NVIDIA-Jetson and RaspberryPi devices and, collect latency and energy consumption metrics.

A.2 Custom OVSF Code Generator

This section includes further details on the implemented OVSF code generator, one of the key components inside `CNN-WGen`. To produce the i-th subtile, the OVSF generator has to feed the compute datapath with ρK^2 OVSF codes. Following TiWGen, the basis vectors are processed in a blocked manner with a tile size of M. But, how can the code generator sustain the processing requirements demanded by `CNN-WGen`? If $M \leq K_i^2$, the M least significant bits (LSBs) are outputted to the compute datapath. If $M > K_i^2$, the code

is self-concatenated $\left\lfloor \frac{M}{K_i^2} \right\rfloor$ times and written to the output's LS part. Simultaneously, the $\text{mod}(M, K_i^2)$ LSBs of the OVSF code are written to the output's MSBs and the constructed vector is passed to the compute datapath. With this approach, when the OVSF codes are read again out of the OVSF FIFO after ρK_i^2 cycles (*i.e.* in the next iteration of the loop on line 2), they are correctly aligned to directly match TiWGen's tiling pattern. In this way, costly selection logic or redundant storage is avoided. A diagram showing these two scenarios for generating OVSF codes is shown in Figure A.1.

A.3 Comparing `unzipFPGA` to existing FPGA designs

Here, we evaluate `unzipFPGA` with diverse existing FPGA designs. In all cases, we use the OVSF50 variant with less than 1-pp accuracy drop. As shown in Table A.1, we compare with Snowflake [51] for ResNet18, the SOTA sparse CNN FPGA design [238] for ResNet34, and [298] that addresses PE underutilisation for SqueezeNet. On Z7045, `unzipFPGA` achieves 2.33× and 1.12× higher throughput than [51] and [238], respectively. For SqueezeNet, our design delivers 1.40× in inf/s/DSP and 1.14×-1.27× in inf/s/-Logic over [298] with the same or 36% less on-chip memory.

ResNet50 Results. The original ResNet50 reaches 76.15% accuracy with 25.56M parameters. Our ResNet50-OVSF50 improves accuracy to 76.23% with 22.83M parameters. Table A.2 presents the comparison for ResNet50. On Z7045, `unzipFPGA` outperforms Snowflake by 1.59× in inf/s. Compared with designs on larger devices, our design achieves higher performance density (GOp/s/DSP) by 3.71×, and 1.76×-6.16× over ResNetAccel and ALAMO, respectively, and consistently higher GOp/s/Logic. FTDL reaches higher GOp/s/DSP and 1.47× lower GOp/s/Logic, but targets a platform with 2.33× larger on-chip memory and 2× higher bandwidth, which substantially reduce the off-chip memory accesses and the associated latency.

A.4 Design Space Exploration for `unzipFPGA`

To estimate the performance and resource usage of different architectural parameters, an analytical modeling framework was developed. This model is used to guide the parameter-

	ResNet18 [51]	unzipFPGA: ResNet18*	ResNet34 [238] DeepCompression	unzipFPGA: ResNet34*	SqueezeNet [298]		unzipFPGA: SqueezeNet*
FPGA	Z7045	Z7045	Z7045	Z7045	V485T	V690T	ZU7EV
Clock (MHz)	250	150	166	150	170	170	200
Precision	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed
DSPs†	900	900	900	900	2800	3600	1728
LUTs	218.6 k	218.6 k	218.6 k	218.6 k	303.6 k	433.2 k	230.0 k
Block RAM	2.40 MB	2.40 MB	2.40 MB	2.40 MB	4.52 MB	6.46 MB	4.75 MB
DSP Util.†	28.4%	100%	56.8%	100%	80%	80%	100%
inf/s	21.38	49.90	27.84	31.1	913.40	1173.00	792.20
inf/s/DSP†	0.0237	0.0576	0.0309	0.0369	0.3260	0.3258	0.4584
inf/s/Logic	0.0978	0.2282	0.1273	0.1422	3.0085	2.7077	3.444

Table A.1: Comparison with existing FPGA work on ResNet18 (4.03 GOps), ResNet34 (7.40 GOps) and SqueezeNet (0.78 GOps).($\star$) using OVSF50, (†) 18×18, 19×18 and 25×18 DSP configurations.

	Snowflake [120]	unzipFPGA: ResNet50*	ResNetAccel [240]	ALAMO [239]		FTDL [300]	unzipFPGA: ResNet50*
FPGA	Z7045	Z7045	Ar10 GX1150	Ar10 GX1150	St10 GX2800	VU125	ZU7EV
Clock (MHz)	250	150	150	240	300	650	200
Precision	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed	16b fixed
DSPs†	900	900	3036	3036	11,520	1200	1728
Logic	218.6 kLUTs	218.6 kLUTs	427.2 kALMs	427.2 kALMs	933.0 kALMs	716.0 kLUTs	230.0 kLUTs
Block RAM	2.40 MB	2.40 MB	6.60 MB	6.60 MB	28.62 MB	11.075 MB	4.75 MB
DSP Util.†	28.4%	100%	56.8%	80%	80%	100%	100%
inf/s	17.7	28.18	33.93	71.38	77.55	151.22	71.71
inf/s/DSP†	0.0196	0.0313	0.0111	0.0235	0.0067	0.1260	0.0415
inf/s/Logic	0.0809	0.1289	0.0794	0.1671	0.0831	0.2112	0.3117

Table A.2: Comparison with existing FPGA work on ResNet50 (8.41 GOps). Ar10 and St10 stand for Arria 10 and Stratix 10, respectively. The Adaptive Logic Modules (ALMs) are Intel's implementation of LUTs. ($\star$) using OVSF50, (†) 18×18, 19×18 and 25×18 DSP configurations.

ization of CNN-WGen given the (1) available memory and compute resources on the target FPGA and (2) the model architecture. In order to efficiently and accurately sample this large search space we must first provide a closed form notation for (1) resource utilization of a candidate design as well as its (2) performance in terms of throughput.

Performance Model. The workload of a CNN with N_L layers is represented as a sequence of $W_i{=}\langle R_i, P_i, C_i\rangle$ *workload tuples* with $i \in \{1, ..., N_L\}$. Given a design point $\sigma{=}\langle M, T_R, T_P, T_C\rangle$, the CNN-WGen's runtime for generating the i-th layer's weights required to compute an $(T_R \times T_C)$ output tile is given by

$$t^i_{\texttt{CNN-WGen}}(\sigma, W_i) = \lfloor \rho \cdot l \rfloor \cdot \left\lceil \frac{T_P \cdot T_C}{M} \right\rceil \cdot \left\lceil \frac{P_i}{T_P} \right\rceil \quad \text{(A.1)}$$

where ρ and l are the OVSF ratio and basis length respectively, M is the number of sub-tiles and, with one factor for each of the pipelined loops in Algorithm 1. With the α

values transferred upfront and the OVSF method generating all weights on-chip, the off-chip memory transfers involve only the input and output activations:

$$t^i_{\text{mem in}}(\sigma, W_i) = \frac{T_R \cdot P \cdot WL}{bw_{\text{in}}}, \quad t^i_{\text{mem out}}(\sigma, W_i) = \frac{T_R \cdot T_C \cdot WL}{bw_{\text{out}}} \tag{A.2}$$

where WL is the adopted wordlength, and $bw_{\{\text{in,out}\}}$ are the memory bandwidths for transferring inputs/outputs.

With the T_C and T_P dimensions unrolled, the computation of an output tile by the accelerator's CNN engine requires the pipelined processing of $\frac{P_i}{T_P}$ tiles for each of the T_R rows. Hence, the CNN engine's runtime for each output tile is estimated as $t^i_{\text{eng}}(\sigma, W_i) = T_R \left\lceil \frac{P_i}{T_P} \right\rceil$. By making use of an input selective PEs, allowing to ease the problem of some PEs being underutilised in layers where the number of output channels is lower than the number of instantiated PEs (i.e. $C \leq T_C$)[1], the runtime is refined as

$$t^i_{\text{eng}^*}(\sigma, W_i) = \left(T_C - C_i + \left\lceil \frac{T_R \cdot C_i - (T_C - C_i) \cdot (C_i + 1)}{T_C} \right\rceil \right) \cdot \left\lceil \frac{P_i}{T_P} \right\rceil \tag{A.3}$$

where the T_R dimension is partially unrolled by processing different rows of T_R through the underutilised PEs. Overall, the accelerator forms a pipeline of three coarse stages: the concurrent input transfer and weights generation, the CNN engine processing and, the output transfer. In this context, the initiation interval[2] of the architecture is given by

$$II^i(\sigma, W_i) = \max\left(\max\left(t^i_{\text{mem in}}, t^i_{\texttt{CNN-WGen}}\right), t^i_{\text{eng}^*}, t^i_{\text{mem out}}\right) \tag{A.4}$$

and $t^i_{\text{total}}(\sigma, W_i) = II^i(\sigma, W_i) \left\lceil \frac{R_i}{T_R} \right\rceil \left\lceil \frac{C_i}{T_C} \right\rceil$ yields the total runtime for the i-th layer. Thus, given a CNN's workload tuple $W = \langle W_i \mid \forall i \in \{1, ..., N_L\} \rangle$, the throughput in inferences per sec (inf/s) is estimated as $T(\sigma, W) = 1 / \sum_{i=1}^{N_L} t^i_{\text{total}}(\sigma, W_i)$.

Resource Consumption Model. From a resource perspective, the main design constraints are the DSPs and on-chip RAM blocks of the target FPGA, or BRAM. Assuming that all MAC operators are mapped to DSPs, the values of $\langle M, T_P, T_C \rangle$ are constrained as

[1] This is one of the contributions presented in [333], which acts as a load-balancing mechanism while only requiring the instantiation of low-overhead additional circuitry. This resulted in upto 20% throughput in compute-bound layers. We omitted providing further details to keep Chapter 3 more focused.

[2] The *initiation interval* is normally defined as the number of cycles required for a function to finish its tasks before accepting a new input.

$D_{\text{MAC}} \times (M + T_P T_C) \leq D_{\text{fpga}}$, with D_{fpga} the available DSPs and D_{MAC} the DSPs/MAC. In our case, 16-bit fixed-point precision is used, where $D_{\text{MAC}}{=}1$ on the evaluated FPGAs.

In terms of on-chip RAM, the accelerator has the I/O and Alpha buffers with wordlength WL and the binary OVSF FIFO, with a total capacity requirement as given by

$$\left(2(T_R T_P + T_R T_C) + D^{\text{Alpha}} N_P^{\text{Alpha}}\right) WL + K_{\text{max}}^2 K_{\text{max}}^2 \leq C_{\text{fpga}} \tag{A.5}$$

where the factor of 2 accounts for double-buffering and C_{fpga} is the on-chip RAM capacity of the target device.

To further estimate the consumption of LUTs, we used a set of place-and-route measurements and fitted linear regression models as a function of `unzipFPGA`'s tunable parameters. Overall, we formally capture the on-chip resource consumption of a design point σ by means of vector $\boldsymbol{rsc}(\sigma)$ that holds the utilised amount of DSPs, BRAMs and LUTs. Similarly, we denote the resource vector of the target platform by $\boldsymbol{rsc}_{\text{Avail.}}$.

Configuration Optimisation. In order to yield the highest performing design for the given CNN-FPGA pair, we cast the design space exploration as a formal optimisation problem: $\max_{\sigma} T(\sigma, W)$ s.t. $\boldsymbol{rsc}(\sigma) \leq \boldsymbol{rsc}_{\text{Avail.}}$. Given a CNN-FPGA pair, we perform exhaustive search, exploring different resource allocations between `CNN-WGen` and the CNN engine. All designs that violate the resource constraints are pruned as infeasible to accelerate the exploration.

A.5 Parametrization of OVSF-based convolutional layers

In Table A.3 we indicate the compression ratios ρ utilized in the experimental evaluation in Section 3.5. These ratios were manually setup and shared for all architectures evaluated for both CIFAR-10 and ImageNet. We empirically observed that deeper lagers can be compressed (i.e. lower ρ) more aggressively without impacting final accuracy. Informed by this observation, we set each OVSF configuration. Naturally the ρ assigned to each convolutional layer becomes yet another hyperparameter to tweak for each architecture, dataset and memory bandwidth supported in each FPGA platform. This was first discussed in Section 3.6 as an immediate extension of the work presented in Section 3.3.

Configuration Name	Block #1 Conv1	Block #1 Conv2	Block #2 Conv1	Block #2 Conv2	Block #3 Conv1	Block #3 Conv2	Block #4 Conv1	Block #4 Conv2
OVSF-100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
OVSF-75	1.0	1.0	0.5	0.5	0.5	0.6	0.6	0.5
OVSF-50	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.4
OVSF-25	1.0	1.0	0.4	0.4	0.25	0.25	0.125	0.125
OVSF-12.5	1.0	0.75	0.25	0.25	0.125	0.125	0.125	0.125

Table A.3: Value of OVSF ratio ρ assigned to each pair of 3×3 convolutional layer in each Residual Block or BottleNeck Block in ReNets. For SqueezeNet's Fire module, which contains a single 3×3 convolutional layer, we make use of the value ρ under the *conv1* columns.

A.6 Winograd-aware Layers for Other Architectures

The results of our study of Winograd-aware networks presented Section 4.5.1 showed multiple configurations of the ResNet-18 architecture at different width-multipliers, bitwidths, quantization levels and convolution algorithms. Here, we present a similar analysis for two other popular architectures for image classification. We limit our study to the full models (i.e. mult=1.0) and show results for SqueezeNet [173] in Table A.4 and for ResNeXt [377] in Table A.5. These results align with what was observed for ResNet-18: In the presence of quantization, learning the Winograd transformations (*flex* configurations) resulted in superior performance than using the default (*static*) transformations. All experiments used the same hyper-parameters as described in Section 4.5.1.

Conv. Type	Bits act. / param.	WA trans.	Accuracy (%) CIFAR-10	Accuracy (%) CIFAR-100
`im2row`	32 / 32	-	91.13	69.06
WA_{F2}		static	91.31	69.42
WA_{F2}		flex	91.25	69.36
WA_{F4}		static	91.23	69.14
WA_{F4}		flex	91.41	69.32
`im2row`	8 / 8	-	91.15	69.34
WA_{F2}		static	90.88	70.06
WA_{F2}		flex	91.03	70.18
WA_{F4}		static	79.28	55.84
WA_{F4}		flex	90.72	69.73

Table A.4: Comparison between standard convolutions (`im2row`) and Winograd-aware layers for SqueezeNet. With INT8 quantization and using the default transformation matrices (*static*), larger tile sizes (i.e. F4) introduce substantial numerical error and result in a sever accuracy drop. This drop in accuracy is significantly reduced if the transformations are learnt (*flex*).

Conv. type	Bits act. / param.	WA trans.	Accuracy (%) CIFAR-10	CIFAR-100
im2row		-	93.17	74.54
WA_{F2}		static	93.19	74.66
WA_{F2}	32 / 32	flex	93.08	74.58
WA_{F4}		static	93.24	74.47
WA_{F4}		flex	93.15	74.62
im2row		-	93.40	74.89
WA_{F2}		static	92.93	75.32
WA_{F2}	8 / 8	flex	93.11	75.80
WA_{F4}		static	76.73	51.20
WA_{F4}		flex	93.29	75.35

Table A.5: Comparison between standard convolutions (`im2row`) and Winograd-aware layers for ResNeXt_20(8×16). With INT8 quantization and using the default transformation matrices (*static*), larger tile sizes (F4) introduce substantial numerical error and result in a sever accuracy drop. This drop in accuracy is significantly reduced if the transformation matrices are learnt (*flex*).

For both architectures, INT8 Winograd-aware $F4$ models with learnt Winograd transformations did not result in a accuracy gaps as pronounced as the ones reported for ResNet-18 in Section 4.5.1. These models even surpass the `im2row` baselines for CIFAR-100. We argue this is because SqueezeNet and ResNeXt_(8×16) have fewer 3×3 convolutional layers (8 and 6, respecitvely) compared to ResNet-18, which has 16. Therefore, the succession of fewer convolutional layers implemented as Winograd convolutions reduces the overall impact of numerical error.

A.7 Overhead of Learnt Winograd Transforms

The default Winograd transformation matrices contain varying amounts of 0's. For $F2$ the sparsity ratios are 50%, 33% and 25% respectively for B^T, G and A^T. These transforms were shown in Equation (4.1). From the construction process of these matrices and specially the choice of *polynomial points*, we would expect lower sparsity ratios as these transforms are adjusted for larger input patches. For example, for the default transforms $F4$ these ratios are 22%, 22% and 25%. For implementations of matrix-matrix multiplications that can exploit data sparsity, as is the case of Arm's Compute Library, having more zeros means less compute which often translate into lower latencies.

The Winograd-aware formulation presented in Chapter 4 does not impose restrictions on how the learnt transform should look like. As a consequence, the resulting transforms

rarely contain zeros. This translates in additional compute for input $B^T dB$ and output $A^T yA$ transforms. The impact of using dense, learnt, transforms for WA_{F4} models running on a Cortex-A73 is a latency increase of 17% (+8ms) and 20% (+6ms) for FP32 and INT8 respectively for a ResNet18 network. This increase in latency is higher on the Cortex-A53 since the Winograd transforms are proportionally more expensive on this core. These penalties represent the worst case performance increase, assuming the transforms are compute bound. However, we believe that due to the access patterns of the Winograd transform kernels (gather and scatter across a wide area of memory) at least some of the performance of the transforms results from misses in the cache hierarchy and so some additional computation can be tolerated without necessarily increasing execution time.

We note that the impact for $F2$ models is considerably higher especially since the original transforms G and A are, not only sparse, but binary and the learnt ones are not. However, these penalties are never met in practice since $F2$ Winograd-aware models with default transforms can perform equally well as those with learnt transforms (as shown in Figure 4.7 and Tables A.4 and A.5) even in INT8. Even with the performance loss due to the learnt transforms, we are still demonstrating some (non-negligible) $1.54\times$ and $1.43\times$ speedup compared to INT8 `im2row` for Cortex-A73 and Cortex-A53 respectively.

A.8 Architectures Optimized with **`wiNAS`**

Our framework `wiNAS`, takes a given macro-architecture and optimizes each 3×3 convolutional layer by choosing from direct convolution or different Winograd configurations. For the search, all 1×1 convolutions were fixed to use `im2row`.

For $\texttt{wiNAS}_{\texttt{WA}}$ in FP32, the resulting architecture only substituted the last convolution layer with `im2row` instead of $F2$. The rest of the layers remained unchanged from the WA_{F4} configuration (which was described in Section 4.5.1). The same micro-architecture was used in CIFAR-10 and CIFAR-100.

For $\texttt{wiNAS}_{\texttt{WA}}$ with 8-bit quantization and CIFAR-10, `wiNAS` replaced the 5th and second last convolutional layers with `im2row`, instead of $F4$ and $F2$ respectively. For CIFAR-100, more optimization was compared to WA_{F4}. The resulting micro-architecture optimization is shown in Figure A.2 (left).

When introducing quantization in the search space, $\texttt{wiNAS}_{\texttt{Q}}$, the resulting architectures are shown in Figure A.2 for both CIFAR-10 (middle) and CIFAR-100 (right).

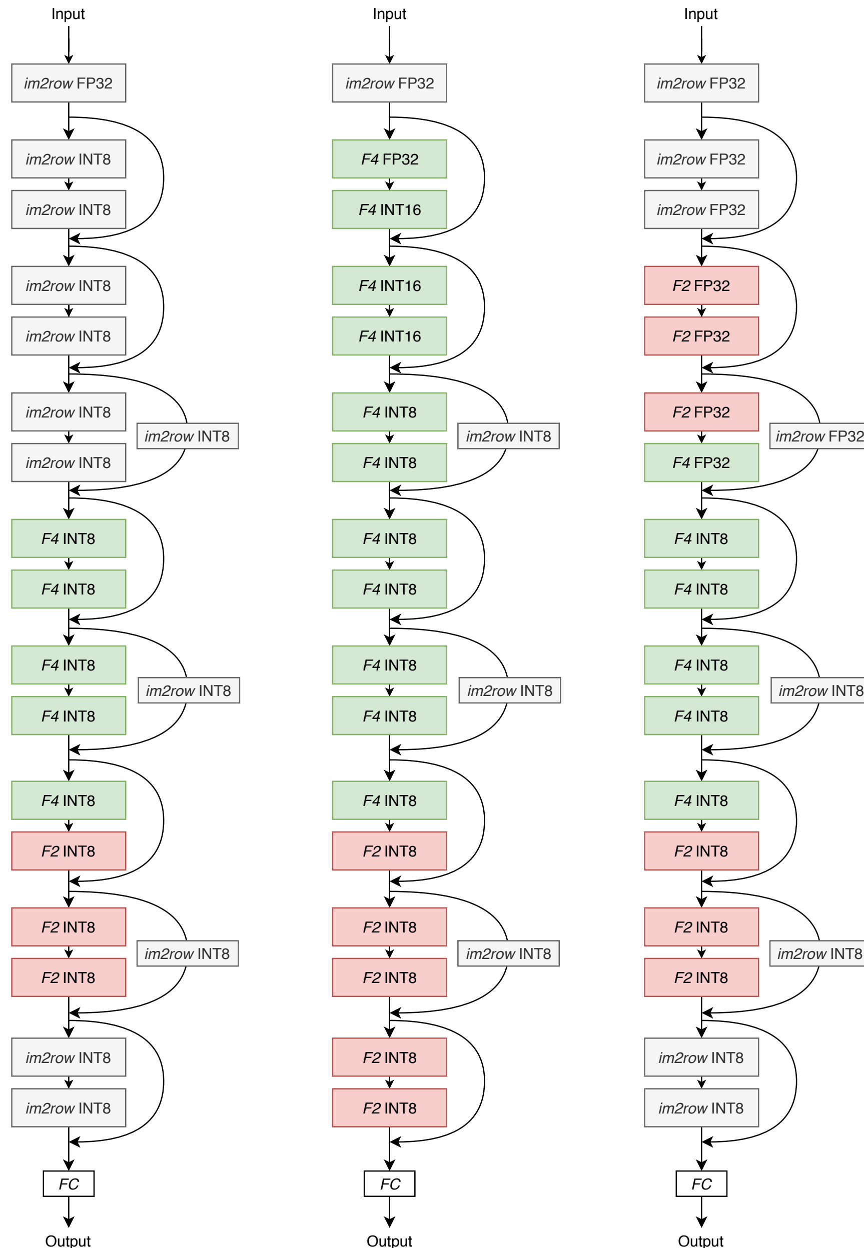


Figure A.2: Resulting architectures after optimizing a ResNet-18 macro-architecture using wiNAS. For wiNAS_{WA} and CIFAR-100, the architecture resulted is shown on the left. With $\text{wiNAS}_{\text{WA-Q}}$, that introduces quantization in the search space, the optimization resulted in different architectures for CIFAR-10 (middle) and CIFAR-100 (right), evidencing the difference in complexity of the latter.

A.9 Deviation of Winograd-aware Layers

In Figure A.3 we show our preliminary analysis attempting to understand what how Winograd-aware layers with learnable transforms differ from standard convolutions. Such analysis was done using a ResNet-18 with multiplier set to 0.25 and, following the same parameter-

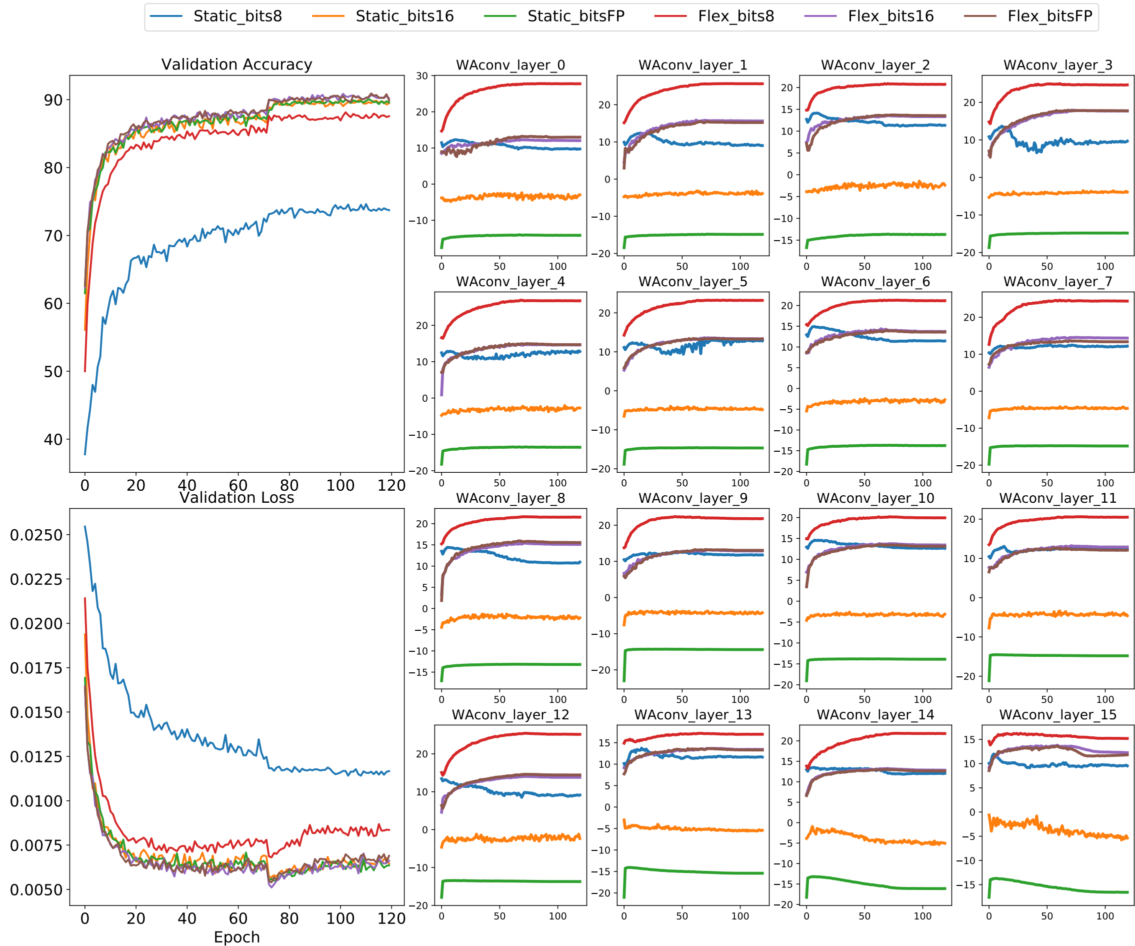


Figure A.3: Deviation in the output of 3×3 Winograd-aware convolutional layers w.r.t standard convolution when using *static* transforms and when using *flex* transforms. We show this difference (measured as MSE) for Winograd-aware layers at INT8, INT16 and FP32. Larger plots of the left show the validation and loss accuracies of each of those layer configurations. As expected, *static* layers perform worse at lower bitwidths. The smaller plots show the MSE difference measured at the output of each Winograd-aware layer compared to that that would had been obtained using the same weights and input but following a standard convolution. The y-axis is in logarithm scale and x-axis in all plots represent the epoch number. The difference is marginal with *static* layers at FP32, as expected. This grows as bitwidth is reduced (due to the numerical errors in Winograds). For *flex* layers, even at FP32 the difference w.r.t normal convolutions is very large and, grows for INT8.

ization as described in Section 4.5.1. The results evidence the large discrepancy in outputs of the convolution given the exact same input and weights tensors and, opens up new possibilities to further modify the Winograd algorithm. We believe this should be possible since, as demonstrated in Chapter 4, even though Winograd-aware layers do not perform a convolution they achieve excellent performance and become the only option to deploy Winograd-convolutions in the context of integer-only arithmetic.

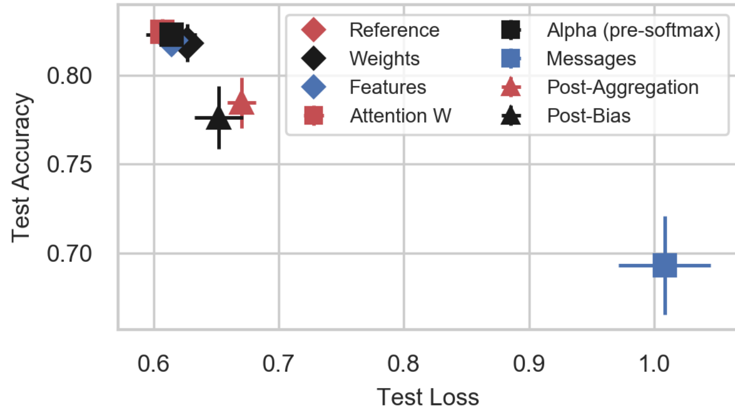


Figure A.4: Analysis of how INT8 GAT performance degrades on Cora as individual elements are reduced to 4-bit precision *without DQ*. For GAT the message elements are crucial to classification performance. Similarly to GIN layers, quantizing the `update` stage results in severe degradation, although not as much as in the former.

A.10 Additional Results for `Degree-Quant`

This section includes additional figures for Chapter 5. In Figure A.4 we show the degradation in node classification accuracy when all stages in a GAT network are quantized with QAT at INT8 and, only one is quantized at INT4 also with QAT. Similar to what was first observed for GIN layers in Figure 5.7, the message generation stage requires accurate representations and heavily suffers from further quantization.

Figure A.5 shows the validation loss curves at different $p_{\min}$ and $p_{\max}$ values with `Degree-Quant`. The resulting accuracies for each configuration are shown next to it in Appendix A.10.

Figure A.7 shows a stage-by-stage diagram of how a direct nQAT is applied to the graph setting. QAT follows the same structure but without stochastic quantization of weights. On the other hand, `Degree-Quant` does include topology-awareness to the quantization process as shown in Figure 5.4. Finally, in Figure A.6 we show the graph-summarization stages for both QAT and `Degree-Quant`. These stages are used for graph classification (i.e. CIFAR-10 and MNIST) and graph regression tasks (i.e. ZINC).

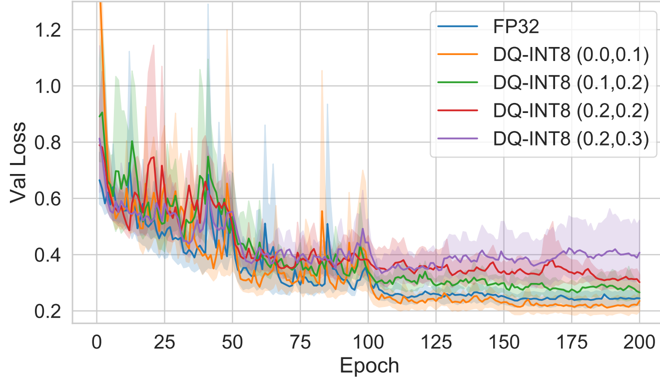


Figure A.5: Validation loss curves for GIN models evaluated on REDDIT-BINARY. Results averaged across 10-fold cross-validation. We show four DQ-INT8 experiments each with a different values for ($p_{\min}$,$p_{\max}$) and our FP32 baseline.

Quantization	Model	REDDIT-BIN ↑
Ref. (FP32)	GIN	92.2 ± 2.3
Ours (FP32)	GIN	92.0 ± 1.5
DQ-INT8 (0.0, 0.1)	GIN	91.8 ± 2.3
DQ-INT8 (0.1, 0.2)	GIN	90.1 ± 2.5
DQ-INT8 (0.2, 0.2)	GIN	89.0 ± 3.0
DQ-INT8 (0.2, 0.3)	GIN	88.1 ± 3.0

Table A.6: Final test accuracies for FP32 and DQ-INT8 models whose validation loss curves are shown in Figure A.5

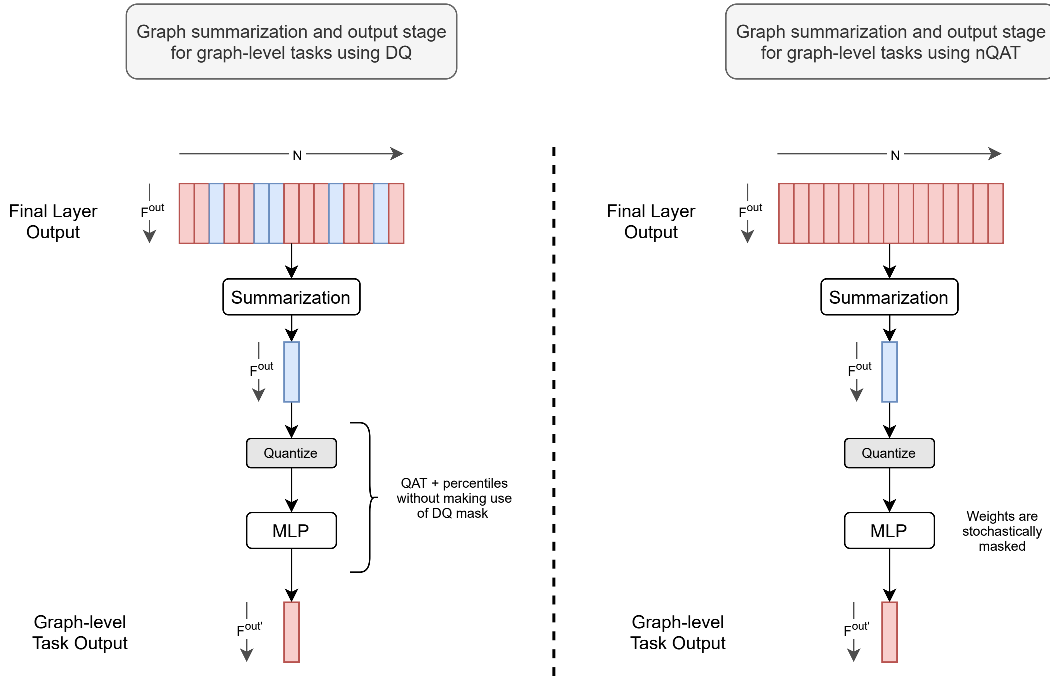


Figure A.6: Diagrams representing how the output graph-summarization stages for graph-level tasks (e.g. graph classification, graph regression) are implemented when making use of DQ (left) and nQAT (right). GNNs making use of DQ during the node-aggregation stages (see fig. 5.4), do not use the stochastic element of DQ in the output MLP layers but still make use of percentiles. For models making use of nQAT, the final MLP still makes use of stochastic quantization of weights.

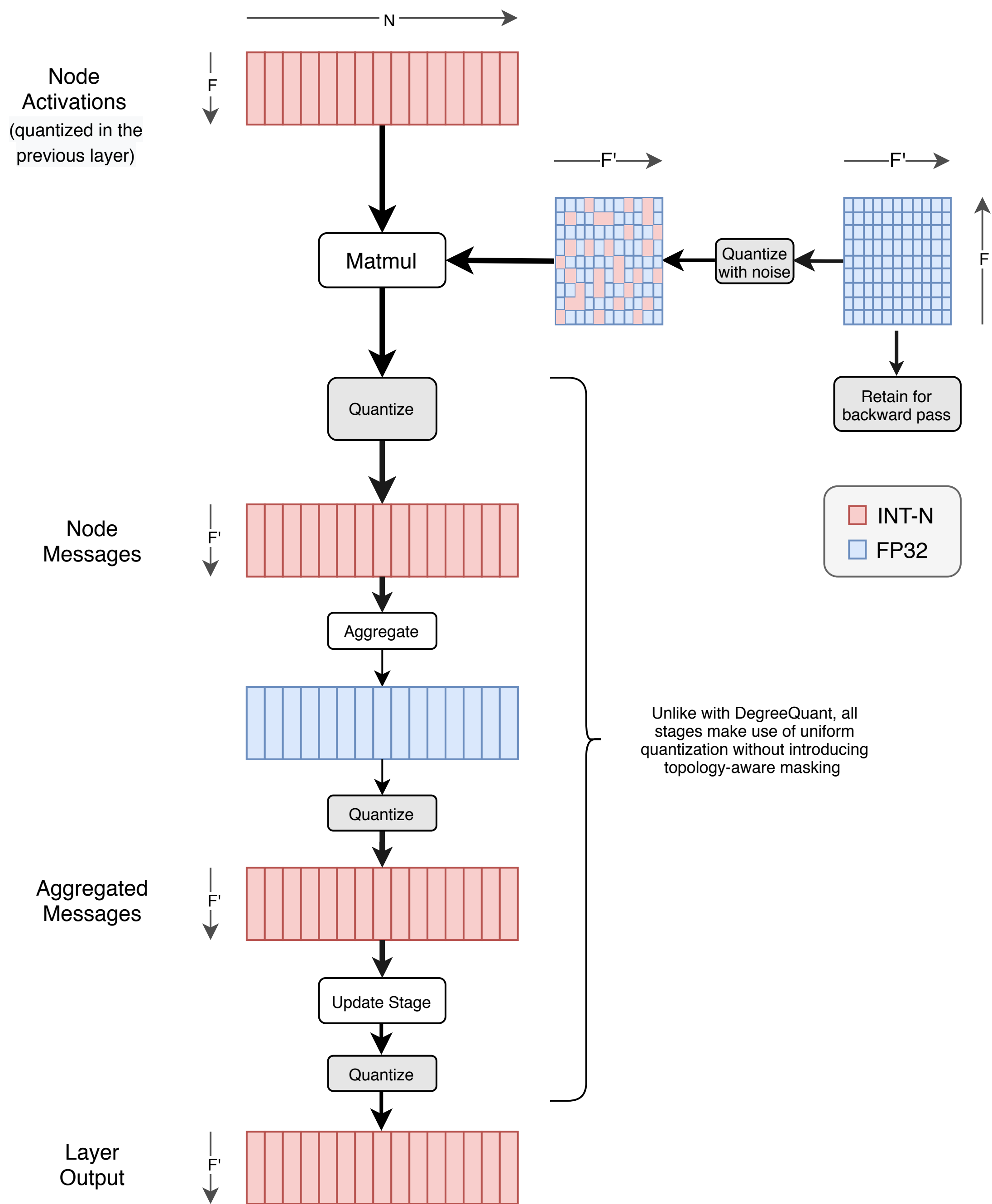


Figure A.7: Diagram representing how nQAT (vanialla QAT with quantization noise for the layer weights as in [309]) is implemented for GNNs. The diagram illustrates this for a GCN layer. The stochastic stage only takes place when quantizing the weights, the remaining of the quantization modules happen following a standard QAT strategy. A QAT diagram would be similar to this one but fully quantizing the weights.