UNIVERSITY OF CAMBRIDGE

Software and Hardware Co-design for Efficient Neural Networks

Yiren (Aaron) Zhao

St Edmund's College

This dissertation is submitted on September, 2021 for the degree of Doctor of Philosophy

Declaration

Yiren (Aaron) Zhao
October, 2021

Abstract

Software and Hardware Co-design for Efficient Neural Networks

Yiren (Aaron) Zhao

Deep Neural Networks (DNNs) offer state-of-the-art performance in many domains but this success comes at the cost of high computational and memory resources. Since DNN inference is now a popular workload on both edge and cloud systems, there is an urgent need to improve its energy efficiency. The improved efficiency enables the use of DNNs in a boarder range of target markets and helps boost performance as performance if often limted by power today. This thesis makes a number of contributions that help improve DNN inference performance on different hardware platforms using a hardware-software co-design approach.

I first show a number of software optimisation techniques for reducing DNN run-time costs. Overall, I demonstrate model sizes can be reduced by up to 34×. A combination of different styles of neural network compression techniques can offer multiplying gains in shrinking the memory footprints. These techniques are suitable for running DNN inference on memory-sensitive edge devices. Using the run-time and data-dependent feature information, I develop a dynamic pruning strategy that outperforms existing static pruning methods by a significant margin. The proposed dynamic pruning not only reduces the model sizes but also the number of multiply-accumulate operations for GPUs. I also introduce a novel quantisation mechanism that is tuned to fit the natural distributions of model parameters and this method decreases the total number of bit-wise operations required for DNN inference.

I then focus on accelerating DNNs using custom hardware. I build a framework named Tomato that generates multi-precision and multi-arithmetic hardware accelerators on FPGA devices. The software hardware co-generation flow deploys hardware accelerators from high-level neural network descriptions, and exploits the hardware reconfigurability of FPGA devices to support a flexible per-layer quantisation strategy. I then demonstrate that the automatically generated accelerators outperform their closest FPGA-based competitors by at least 2 to 4× for latency and throughput. The accelerator generated for the ImageNet classification runs at a rate of more than 3000 frames per second with a latency of only

$0.32ms$, making it a suitable candidate for latency critical, high throughput inference in the cloud.

Finally, I show how automated machine learning techniques can be improved with hardware-awareness to produce efficient network architectures for emerging types of neural networks and new learning problem setups. Hardware-aware network architecture search (NAS) is able to discover more power efficient network architectures and achieve significant computational savings on emerging neural networks types such as graph neural networks. The proposed Low Precision Graph Network Architecture Search improves the size-accuracy Pareto frontier when compared to seven manual and NAS-generated baselines on eight different graph datasets. In addition, I demonstrate hardware-aware NAS can be applied to a *many-task many-device* few-shot learning scenario. In popular few-shot learning benchmarks with various hardware platforms and constraints, the proposed approach outperforms a variety of NAS and manual baselines by a significant margin. On the 5-way 1-shot Mini-ImageNet classification task, the proposed method outperforms the best manual baseline by 5.21% in accuracy using 60% less computation.

Acknowledgements

Many say pursuing a PhD is a lonely endeavor, throughout my PhD, I have recieved a great deal of support and this journey then becomes an unforgettable and fun one for me.

I will start by acknowledging and giving my warmest thanks to my supervisor, Prof. Robert Mullins. His support went far beyond looking at our paper drafts and discussing research ideas. I felt I have been increditably lucky to have him as my PhD supervisor and will always be truely thankful to the amount of reserach freedom that he lets me to have in my PhD studies.

I would also thank my PhD examiners, Prof. Nic Lane and Dr. Elliot Crowley, for their fruitful discussion and insightful questions during the viva. These suggestions have definitely helped me to improve my thesis.

I would also give special thanks to my parents (Mr. Qingjie Zhao and Mrs. Shihong Li) and my partner (Ms. Shu Zhong). I feel lucky to be in a super supportive family, and I am grateful to my parents for their continuous support and love. I could not achieve any of these without your support.

I owe speical thanks to all my reserach collaborators, both in and out of the Computer Lab in Cambridge. I will always remember the time and effort that they put into these fun research projects. I also owe a lot of thanks to all my internship mentors over various summers. I also give my deepest thanks to all my friends, you know who you are. I would not have got through it without you.

Contents

Chapter 1

Introduction

1.1 Motivation

Deep Neural Networks (DNNs) are now a popular technique for solving problems in many fields. Previously, building an object detector or a dialogue system required various sophisticated components and hand-crafted feature extractions from human experts. With the success of DNNs, these complicated building blocks are replaced by a single or multiple learning systems [72, 93]. Since many real-life problems can be viewed as learning from labelled or unlabelled data, the application of DNNs is now popular in domains like natural language processing [162], computer vision [93], decision making [121], autonomous driving [138], *etc.* DNNs are not only broadly adopted in computer science but also show potential in many other fields for accelerating fundamental scientific discoveries, *e.g.*, DNNs have found a totally new antibiotic [154] and have been used to successfully predict the structure of a vast number of proteins from their amino acid sequences [141]. From the hardware perspective, neural networks are becoming a major workload on both power-sensitive embedded devices and large distributed cloud systems.

The rise of DNNs helped researchers to achieve high accuracy on many datasets and showed some interesting trends in terms of their computation requirements. AlexNet, an early model designed by Krizhevsky *et al.* for the 2012 ImageNet competition, trained on 1.2 million images and was used for classification with 1000 categories [93]. The 1000 classes include every-day objects (cat, dog, leopard, *etc.*). The proposed AlexNet utilises around 60 million parameters and around 1.6 billion floating-point operations (FLOPs) for a single inference run. The GPT2 language model, proposed in late 2019, is trained to perform next word prediction but can be further fine-tuned to many downstream language tasks. The model uses around 1.5 billion parameters and 3.4 trillion FLOPs to produce a single inference result and was trained on around 8 million documents. Machine learning is tackling harder and bigger problems and is using more and more data. This in turn requires much larger networks (*e.g.* GPT2 is around 25$\times$ larger than AlexNet) and computational complexity (*e.g.* GPT2 requires around 2000× more computations compared to AlexNet), posing an enormous challenge of executing these workloads on today's or future hardware systems.

The fast-growing computational requirements of Machine Learning (ML) applications pose a real challenge to existing hardware systems. The possibility that hardware may limit what ML is able to achieve, *e.g.* due to power, computation speed or memory bandwidth, is obviously a concern. The deep learning community faced the second artificial intelligence winter in the late 90s. A major reason for the recession was the insufficient computational power since 1) computers at that time were not explicitly designed for massively parallel workloads such as neural networks, and 2) the CMOS technology (*e.g.* number/speed/power of transistors) at that time posts a fundamental limit on the computational capabilities. Machine Learning algorithms, consequently, do not have a chance to show their great performance on large-scale problems. It was a great lesson to learn that when hardware design and their technology scaling cannot keep up with computational requirements, novel research is not easy to conduct and the whole field is slow-moving because of the computation bottleneck. Any advances in terms of efficiency or ML hardware, both from the hardware architecture design and CMOS scaling, will allow us to apply ML methods to a broader range and more complex problems.

In the meanwhile, challenges also exist in terms of technology scaling, it is reaching or may have reached the post Moore's law era [140], the speed and capability of our computers are not growing as fast as they used to be. Since transistors are getting smaller and smaller with CMOS scaling, transistor density grows but then power and wire delays become another major limiting factor [43]. In addition, CMOS scaling has also slowed down significantly compared to the past decade. From the hardware architecture design point of view, enabling the ever-increasing computation requirements of running more and more demanding ML workloads rely on inventions on smart hardware architectures. From the system design's point of view, future systems are going to be heterogeneous, and these systems will have to reach a better performance with a fixed power budget. Future systems would require an integration of various specialised hardware and ML accelerators are expected to play a major role in such a system. Given the growing need of applying Deep Learning to different applications and the potential limitation of CMOS scaling, this dissertation aims at exploring possible ways of reducing the computational requirements of modern DNNs.

1.2 Research questions and hypotheses

In this dissertation, I address the following research questions and hypotheses:

- An efficient combination of algorithmic optimisations makes neural networks more hardware friendly with a given accuracy budget.

- A co-design of neural network architectures, algorithmic optimisations and hardware architectures helps to reach the full potential of DNN acceleration on custom hardware.
- Designing accelerators for neural networks is expensive and repetitive, is it possible to have an automated framework to generate accelerators for custom hardware?
- Given the increase in the number of distinct types of neural networks (*e.g.* convolutional neural network, recurrent neural network, graph neural network, *etc.*) and different learning setups (*e.g.* few-shot learning, federated learning, *etc.*), is it possible to have a hardware-aware Network Architecture Search framework for emerging networks or new learning setups?

1.3 Contributions

This dissertation makes a number of contributions in the field of efficient deep neural network inference.

First, I create and evaluate a range of network compression techniques. These methods are implemented with an open-source software framework (Mayo). The implemented compression techniques reduce either the number of parameters or the number of bits required to represent parameters of a neural network. The open source framework allows users applying different compression techniques to arbitrary tensor components in a neural network, this simplifies the evaluation of different styles of network compression techniques [201, 126]. During the development of the framework, I also demonstrate that a combination of compression techniques normally significantly outperforms any one technique alone, offering a large combinatorial design space for compression techniques.

Second, a novel run-time pruning technique that reduces the computing cost of convolutions with minimal neural network architectural modification. The novel pruning explores the feature saliency at run-time and is so called Feature Boosting and Suppression (FBS) [50]. This early work in dynamic computation of deep neural networks later encouraged exploration of more algorithmic optimisations focusing on the same problem [77, 165, 167] and also CNN hardware accelerators supporting run-time channel skipping [76].

Third, an efficient distribution-aware quantisation and a new general framework of quantising sparse neural networks based on the minimum description length principle [202]. This quantisation illustrates that custom hardware has the ability to produce a better size-accuracy tradeoff by exploring the more flexible hardware operators. In addition, an efficient combination of different compression techniques (pruning and a custom quantisation) can significantly improve the compression performance.

Fourth, an automated framework for generating multi-precision and multi-arithmetic accelerators on custom hardware [203]. Here, instead of using a single large systolic array core as in most neural network accelerators, I exploit a deeply pipelined streaming architecture with multiple small cores that has the advantage of not temporally sharing computing elements for different neural network layers. The streaming processing design offers a flexibility on the arithmetic space since now the hardware can have different layer-wise quantisations. This framework is an example case where if the hardware provides enough flexibility, its corresponding neural network model can have more design freedom and thus a higher re-trained accuracy.

Fifth, a hardware-aware Network Architecture Search (NAS) flow for graph neural networks [205, 204]. The proposed NAS flow not only is gradient-based that joins the normal Stochastic Gradient Descent but also is hardware-aware so that it can produce highly quantised graph neural network models. The NAS-generated networks can be viewed as a result of a joint-optimisation for both model architectures and quantisation choices. The proposed NAS flow is a scalable approach for future new types of neural networks that are going to be deployed with critical model size and latency constraints.

Finally, a general NAS framework for *many-task many-device* few-shot learning. The *many-task many-device* scenario considers deployments of neural network models to different learning tasks (normally networks of the same style but operate on different datasets), and deployments of these models on various hardware systems with different run-time constraints (*e.g.* latency, storage size, *etc.*). I then use a classic few-shot learning setup as a testbed. The empirical results demonstrate that the proposed NAS framework can produce optimised models for each task-device pair and it outperforms manually designed baselines by a significant margin.

The contributions of this dissertation cover several important aspects of accelerating DNN inference. The first three contributions focus on the software stack, these techniques reduce the redundancies of neural networks for more energy efficient inference. However, this dissertation reveals that software compression methods show varying performance on different hardware systems, there does not exist a one-fits-all algorithmic solution for accelerating DNNs on a diverse set of hardwares. In addition, these contributions also demonstrate that the best practice is often applying a series of DNN compression techniques instead of using a single one, there is a multiplying gain when applying several compression techniques together. The fourth contribution shows how flexible, reconfigurable, hardware allows a large, flexible software compression design space to be considered and then supported through a specialised implementation, this HW/SW co-design generates highly efficient CNN accelerators. This contribution demonstrates that detailed hardware architectural choices will influence algorithmic compression methods, and possibly increase the degrees of freedoms of the algorithmic optimisation space. In addition, it demonstrates how deeply pipelined streaming architecture has a latency advantage compared to other systolic-array based architectures. Since the software compression design space is large and complex, the final two contributions focus on automating the process of designing efficient DNNs with hardware-awareness. These contributions demonstrate that NAS can reduce the amount of software tuning significantly. In addition, NAS is a scalable approach for emerging neural networks types (Graph Neural Networks) and new learning setups (*many-task many-device* learning).

1.4 Outline

The organisation of this dissertation is as follows:

In Chapter 2, I discuss the backgrounds of neural networks and existing techniques used in software and hardware for minimising their required computation resources. I will also illustrate the importance and introduce the key challenges for hardware efficient neural network inference.

In Chapter 3, I show that neural network models are inherently redundant and algorithmic compression methods can focus on exploring these redundancies. I introduce the software framework Mayo and show how combining various network compression methods can explore redundancies in different design spaces. In addition, I present two novel compression methods for faster inference. The first one is a dynamic pruning algorithm that can further reduce the computational requirements compared to static pruning. The second novel compression is a distribution-aware quantisation that can help sparse models to be represented using a more compact data type.

In Chapter 4, I focus on the hardware implementation of a convolutional neural network accelerator. I show how to use a streaming-based computation pattern to allow better flexibility in the arithmetic design space. The flexible arithmetic designs then provides the opportunities for the network to achieve a better accuracy. I then describe how I extend this idea to an automated flow for generating multi-precision, multi-arithmetic accelerators on custom hardware.

In Chapter 5, I introduce the concept of Automated Machine Learning (AutoML) and Network Architecture Search (NAS). I demonstrate how they can be applied on graph neural networks and many-task many-device few-shot learning. I then explain how NAS can be a scalable solution for emerging network types and new learning setups.

In Chapter 6, I conclude the dissertation and provide suggestions for possible future research on accelerating neural network inference.

1.5 Publication

Some of the research discussed in this dissertation also appeared in the following publications:

- Yiren Zhao, Xitong Gao, Robert Mullins, and Chengzhong Xu. Mayo: A framework for auto-generating hardware friendly deep neural networks. In *Proceedings of the 2ndInternational Workshop on Embedded and Mobile Deep Learning, pages 25–30, 2018. (EMDL 2018).*

- Xitong Gao, Yiren Zhao, Lukasz Dudziak, Robert Mullins, Chengzhong Xu. Dynamic channel pruning: Feature boosting and suppression. In *International Conference of Learning Representations 2019 (ICLR 2019).*

- Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. Focused quantization for sparse CNNs. In *Advances in Neural Information Processing Systems,pages 5584–5593, 2019 (NeurIPS 2019).*

- Milos Nikolic, Mostafa Mahmoud, Andreas Moshovos, Yiren Zhao, and Robert Mullins. Characterizing sources of ineffectual computations in deep learning networks. In *IEEE International Symposium on Performance Analysis of Systems and Software, pages 165–176. IEEE, 2019 (ISPASS 2019).*

- Yiren Zhao, Xitong Gao, Xuan Guo, Junyi Liu, Erwei Wang, Robert Mullins, Peter YK Cheung, George Constantinides, and Cheng-Zhong Xu. Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs. In *2019 International Conference on Field-Programmable Technology, pages 45–53. IEEE, 2019 (ICFPT 2019).*

- Yiren Zhao, Duo Wang, Xitong Gao, Robert Mullins, Pietro Lio, and Mateja Jamnik. Probabilistic dual network architecture search on graphs. In *Deep Learning on Graphs: Method and Applications Workshop for 35th AAAI Conference on Artificial Intelligence (DLG-AAAI 2021), recipient of the best student paper award.*

- Yiren Zhao, Duo Wang, Daniel Bates, Robert Mullins, Mateja Jamnik, and Pietro Lio. Learned low precision graph neural networks. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys 2021).*

- Yiren Zhao, Xitong Gao, Ilia Shumailov, Nicolo Fusi, Robert Mullins. Rapid Model Architecture Adaption for Meta-Learning. In *submission*

During my PhD study, my research also resulted in the following publication and patents that are not included in this dissertation:

- Yiren Zhao, Ilia Shumailov, Robert Mullins, Ross Anderson. To compress or not to compress: Understanding the interactions between adversarial attacks and neural network compression. *Proceedings of the 2nd SysML Conference, 2019 (SysML 2019).*

- Kaifeng Wang, Xitong Gao, Yiren Zhao, Xingjian Li, Dejing Dou, Cheng-Zhong Xu. Pay Attention to Features, Transfer Learn Faster CNNs. *International Conference of Learning Representations 2020 (ICLR 2020).*

- Ilia Shumailov, Yiren Zhao, Robert Mullins, Ross Anderson. Towards Certifiable Adversarial Sample Detection. *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security, Pages 13–24 (AISec 2020).*

- Yiren Zhao, Ilia Shumailov, Robert Mullins, Ross Anderson. Blackbox attacks on reinforcement learning agents using approximated temporal information. *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2020).*

- Daniel Lo, Amar Phanishayee, Eric S Chuang, Yiren Zhao, Richie Zhao. Neural network activation compression with narrow block floating-point. US Patent.

- Daniel Lo, Amar Phanishayee, Eric S Chuang, Yiren Zhao, Richie Zhao. Neural network activation compression with outlier block floating-point. US Patent.

- Daniel Lo, Amar Phanishayee, Eric S Chuang, Yiren Zhao. Neural network activation compression with non-uniform mantissas. US Patent.

- Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, Ross Anderson. Sponge Examples: Energy-Latency Attacks on Neural Networks. . *Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P 2021)*

- David Khachaturov, Ilia Shumailov, Yiren Zhao, Nicolas Papernot, Ross Anderson. Markpainting: Adversarial Machine Learning meets Inpainting. *The 38th International Conference on Machine Learning (ICML 2021, Short Talk).*

- Ilia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Murat A Erdogdu, Nicolas Papernot, Ross Anderson. Manipulating SGD with Data Ordering Attacks. To appear in *Advances in Neural Information Processing Systems, 2021 (NeurIPS 2021).*

Chapter 2

Background

This chapter surveys the background literature and recent model design trends of deep learning. I then use convolutional neural networks as an example to introduce the basic structure and learning methods of neural networks. Later, I introduce popular neural network compression techniques and prior automated machine learning algorithms. At the end, I provide a survey of prior hardware accelerators and their corresponding hardware optimisation.

2.1 An overview of deep neural networks and their computational requirements

In recent years, we see a trend of using larger and larger neural networks to solve an array of tasks with ever-increasing complexities, and all of this started from the early discoveries in the field of computer vision. The design of AlexNet, utilising around 60 million parameters and 1.6 billion floating-point operations (FLOPs), showed a significant accuracy boost compared to manual feature engineering when classifying the 1000 daily objects in the ImageNet competition. This work then motivated researchers to build large neural network models for different learning problems [93]. Simonyan *et al.* and Zagoruyko *et al.*, in the following years, used deeper CNNs with more parameters and achieved significantly better results on the same dataset [151, 188]. Also very recently, in the world of natural language processing (NLP), Vaswan *et al.* proposed to use the attention mechanism to solve language understanding tasks and built the Transformer model with around 213 million parameters [162]. Devlin *et al.* later scaled the transformers to be both bidirectional and deeper and called their model BERT [38]. Very recently, we have begun to see the rise of gigantic models such as GPT2 and GPT3 [133, 15]. These models not only solve traditional NLP tasks, such as machine translation and question answering; they aim adding machine intelligence to the broader range of applications, from translating English to SQL to plotting beautiful graphs from plain English. The GPT3 model uses more than 170 billion parameters, and its training costs around 4.6 million dollars if you are using Azure services [15].

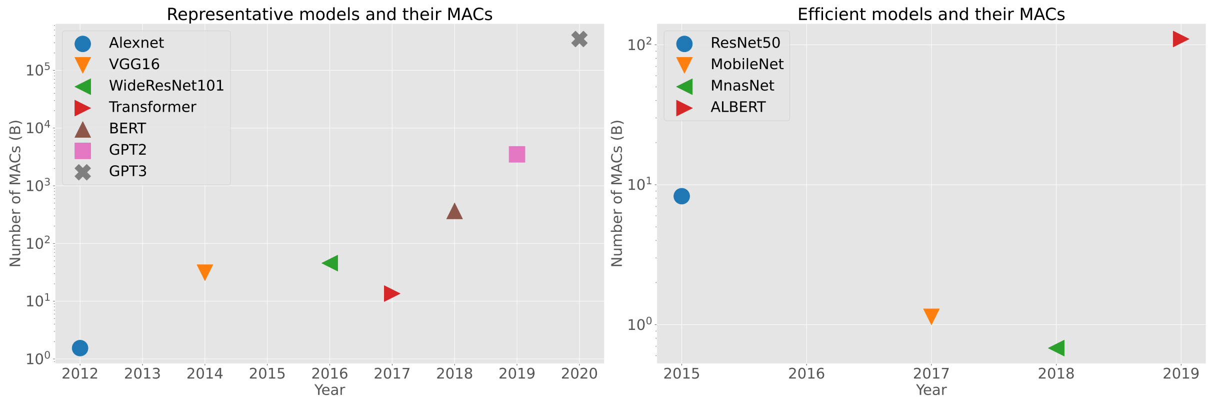


Figure 2.1: Model sizes and numbers of floating points of representative neural networks published in different years. These models have various accuracies and target different tasks, however, the general trend suggests that models are getting bigger and more computationally heavy.

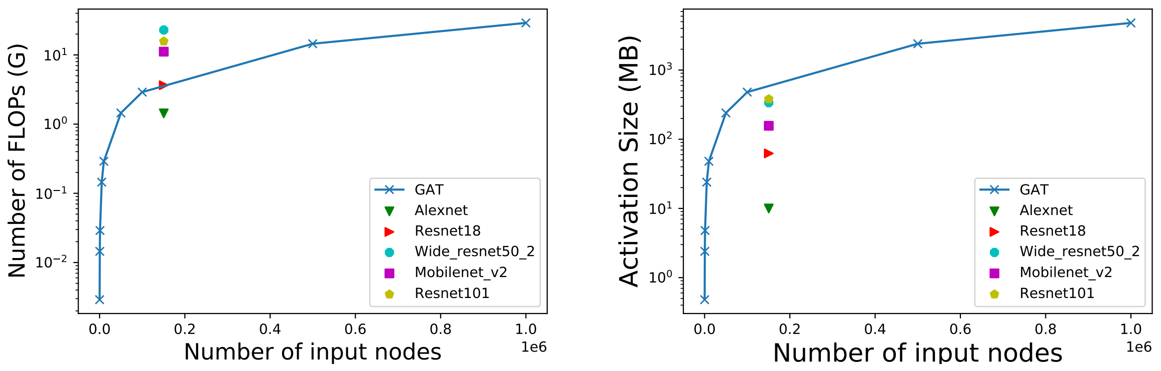


Figure 2.2: Evaluating the computational requirements for a two-layer Graph attention network (GAT) compared to a range of vision models. Numbers are reported for a single graph/image inference and we treat an image as a graph with 224 × 224 × 3 input nodes. Inputs for the GAT model are Erdős-Rényi-Gilbert Random Graphs generated with different number of nodes (N) and a connectivity probability of $\frac{1}{N}$.

While machine learning models are growing in size and complexity, smart network architectural engineering reduces the cost of running inferences. He *et al.* and Howard *et al.* redesigned the building blocks of CNNs, and proposed the ResNet and MobileNet families respectively [66, 73]. These vision models use far fewer parameters and operations, but match the performance of their larger counterparts. Similarly, the design of ALBERT shows the possibility to port a lightweight transformer model with comparable performance [97]. In parallel to manual optimisation of networks based on domain expertise, automatic exploration methods such as Network Architecture Search (NAS) successfully find efficient models such as the MNasNet that reaches the state-of-the-art accuracy per operation performance [157]. In Figure 2.1, I choose the well-known neural networks mentioned above and show how their computational requirements change across years. I demonstrate their computational requirements in terms of 1) number of parameters, also known as model sizes, and 2) number of multiplication-accumulates (MACs) required for a single inference run. The first gives an indication of the hardware memory requirements and the potential need for high off-chip memory bandwidth depending on the detailed memory hierarchy designs. The latter challenges the computational ability of the hardware. The number of parameters of various networks are measured in millions (M), and the number of MACs reported are in billions (B). I also label the names and the published dates of these networks in Figure 2.1. On the same dataset, researchers spent a long time on reducing both the number of parameters and error rates of DNNs. It is noteworthy that efficient networks normally utilise special network structures and optimization techniques. For example, MobileNet made use of grouped convolutions [73] and ResNet50 has its special shortcut connections [66]. There are several interesting trends we can summarise from Figure 2.1:

- It takes a great amount of time to design efficient neural networks. AlexNet was introduced in 2012 but the efficient MobileNet family starts to appear in late 2017.
- People tend to first design networks for better accuracy, but these networks are normally harder to train and have a risk of being more sensitive to hardware. For instance, it is easier to support VGG16 in today's TPUs [85] than WideResNet101 because of the skip connections.

Apart from the ever increasing model sizes and complexities, recent years have witnessed a more diverse set of neural network types. In addition to classic learning domains such as computer vision and natural language processing, emerging research has proven the capabilities of DNNs on graph [89, 163] or tabular data [183]. Novel neural network types often show vastly different computational properties. Taking graph neural networks as an example, Figure 2.2 demonstrates that these models can have a much higher computational requirements in terms of FLOPs and activation sizes compared to vision models if the input graph size is large. However, these graph models are often very small in terms of model sizes compared to their vision counterparts.

In general, we can summarize the following trends:

- We will always have larger datasets and larger model. There will always be a demand to tackle harder problems. The computational requirement is always growing.

- Although over-parameterization is advantageous in training on very large datasets, it is not always necessary for accurate inference [28]. There exists efficient networks on the same datasets, smart architectural changes or automated machine learning will discover the efficient networks in a few years from the release date of the dataset.
- The operations and structures of DNNs will be more diverse and complex (residual blocks, depthwise separable convolutions, attention mechanisms, *etc.*) over time in order to make these models more efficient (*e.g.* run faster or consume less energy for a given accuracy budget).
- Over time new neural network types are introduced to improve performance or solve new machine-learning problems. These may have very different memory access patterns and computational requirements when compared to existing models.

2.2 Deep neural network structures

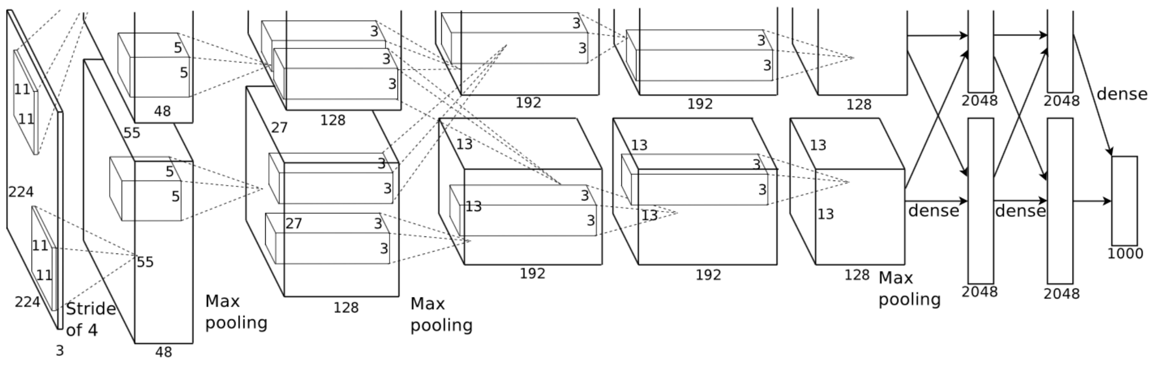


Figure 2.3: AlexNet network architecture with 5 convolutional layers and 3 fully-conneted layers [93].

DNNs have been biologically inspired by the neural system in human brains. Neurons are basic units in a DNN and a number of neurons make up a layer. The network architecture of AlexNet is shown in Figure 2.3. There are currently a great number of DNN layer types, the network in Figure 2.3 utilised fully-connected layer, pooling layer and convolutional layer. AlexNet has 5 convolutional layers, 3 max pooling layers and 3 fully connected layers. The convolutional layers have most of the computations and they start to replace the fully-connected layers because of their capabilities on feature extractions [73]. The fully connected layer is nothing more than dense matrix multiplications and the pooling layer requires only a linear scan of all variables in the feature maps. Since convolutions is a popular and representative computation layer in today's DNNs, I would like to introduce the mathematics behind convolutions for a concrete understanding of this workload.

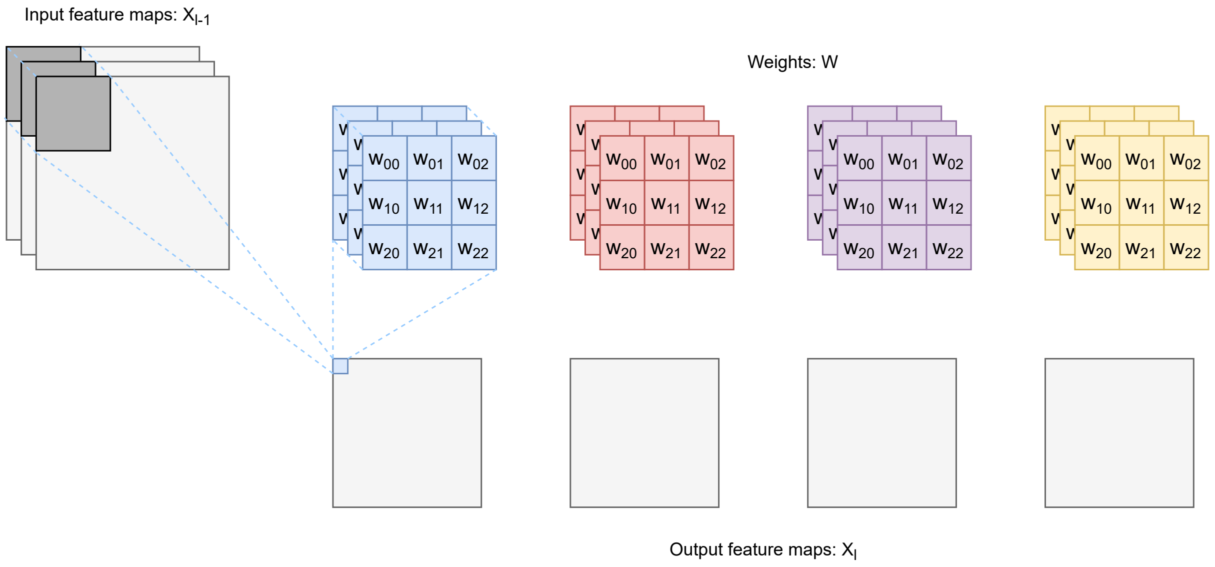


Figure 2.4: A typical convolution layer, with 3 input feature maps, 4 output feature maps and 12 3×3 kernels.

We consider a deep sequential CNN with L convolutional layers, *i.e.* $\mathbf{X}_L = F(\mathbf{X}_0) = f_L(\cdots f_2(f_1(\mathbf{X}_0))\cdots)$, where the l^{th} layer $f_l : \mathbb{R}^{C_{l-1}\times H_{l-1}\times W_{l-1}} \to \mathbb{R}^{C_l\times H_l\times W_l}$ computes the features $\mathbf{x}_l \in \mathbb{R}^{C_l\times H_l\times W_l}$, which comprise of C_l channels of features with height H_l and width W_l. The l^{th} layer is thus defined as:

$$\mathbf{X}_l = f_l(\mathbf{X}_{l-1}) = \mathsf{ReLU}(\mathsf{conv}_l(\mathbf{X}_{l-1}, \boldsymbol{\theta}_l)) \tag{2.1}$$

Here, $\mathsf{ReLU}(x) = \max(x, 0)$ denotes the ReLU activation, and $\mathsf{conv}_l(\mathbf{X}_{l-1}, \boldsymbol{\theta}_l)$ computes the convolution of input features X_{l-1} using the weight tensor $\boldsymbol{\theta}_l \in \mathbb{R}^{C^l\times C^{l-1}\times k^2}$, where k is the kernel size. As illustrated in Figure 2.4, for each single output pixel at a particular channel of the output feature map, the entire weights volume is multiplied with the selected input feature volume in an element-wise manner and then summed to produce the final single pixel result. At inference time, a convolution uses $k^2C_{l-1}C_lH_lW_l$ *multiply-accumulate operations* (MACs), which means $2k^2C_{l-1}C_lH_lW_l$ floating-point operations (FLOPs), for the computation of the l^{th} layer.

Another view on this convolution workload is to treat it as a classic hardware acceleration for nested loops. Listing 2.1 further shows the code snippet for a single convolution. OutR, OutC and OutF are the three dimensions of the output feature maps. InF is the number of input feature maps and K and stride are the kernel sizes and stride value respectively.

Listing 2.1: Nested loops for convolution

```
 for (row=0; row<OutR; row++){
   for (col=0; col<OutC; col++){
     for (fo=0; fo<OutF; fo++){
       for (fi=0; fi<InF; fi++){
         for (i=0; i<K; i++){
           for (j=0; j<K; j++){
             output_fm[fo][row][col]+=weights[fo][fi][i][j]*in_fm[fi][
   stride*row+i][stride*col+j];
}}}}}}
```

Optimisation for computing nested loops has been a long-existing research subject in hardware acceleration. The formulation above offers many possible optimisations for better data-reuse patterns or better contiguous DRAM access patterns [137, 103, 124], these optimisation are beyond the scope of this dissertation. A naive strategy is to unroll all iterators in the nested loops so that all computations are fully parallelised, however, such designs normally cannot meet the hardware constraints due to the large computational dimensions (*e.g.* large channel counts or large feature map sizes) of neural networks. In Chapter 4, I will revisit different optimisation choices for nested loops and how they influence the hardware designs.

2.3 Neural network compression and automated machine learning

Neural network compression refers to the process of reducing the run-time costs of neural network inference through shrinking the network parameters, it normally targets a pre-trained neural network model with a pre-defined network architecture. Reducing the size of a neural network directly decreases the amount of off-chip accesses and normally reduces the amount of computation required. Although re-designing the original network to a smaller network is directly beneficial, such design normally requires a large amount of design-time. In general, most compression techniques focus on the following three paths.

- Reduce the number of parameters in a neural network.
- Reduce the bit-width of parameters.
- Reshape kernels.

I summarise all the compression techniques in Figure 2.5 as a taxonomy. In this section, I will mainly focus on pruning and quantisation. In addition, I will introduce optimisation on the neural network architecture space, namely neural network architecture search techniques for finding the best performing network architectures.

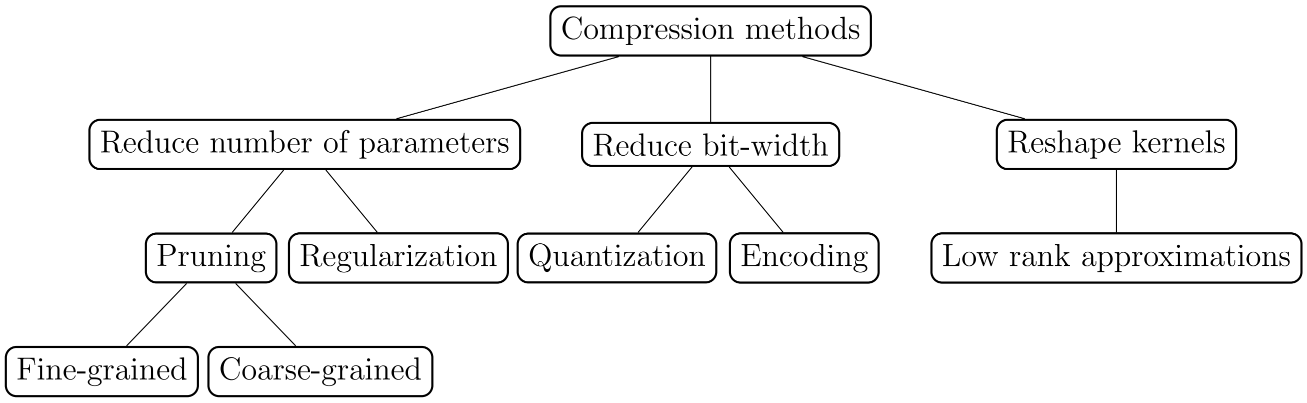


Figure 2.5: Taxonomy of compression techniques.

2.3.1 Pruning

Pruning directly reduces the number of connections. Since LeCun *et al.* [100] introduced optimal brain damage, the idea of creating more compact and efficient CNNs by removing connections or neurons has received significant attention. Early approaches to pruning deep CNNs zeros out individual weight parameters [65, 57]. This results in highly irregular sparse connections, which were notoriously difficult for SIMD architectures such as GPUs to exploit. This has prompted custom accelerator solutions that exploit sparse weights [128, 63]. Although supporting both sparse and dense convolutions efficiently normally involves some compromises in terms of efficiency or performance.

Because of the above-mentioned reasons, recent research is more focused on *structured sparsity* [169, 185, 6, 208], that can be directly exploited by GPUs and also allow custom accelerators to focus solely on efficient dense operations. Wen *et al.* [169] added group Lasso on channel weights to the model's training loss function. This has the effect of reducing the magnitude of channel weights to diminish during training, and remove connections from zeroed-out channels. To facilitate this process, Alvarez *et al.* [6] additionally used proximal gradient descent, while Li *et al.* [105] and He *et al.* [69] proposed to prune channels by thresholds, *i.e.* they set unimportant channels to zero, and fine-tune the resulting CNN. The objective to induce sparsity in groups of weights may present difficulties for gradient-based methods, given the large number of weights that need to be optimised. A common approach to overcome this is to solve [70] or learn [114, 185] channel saliencies to drive the sparsification of CNNs. He *et al.* [70] solved an optimization problem which limits the number of active convolutional channels while minimising the reconstruction error on the convolutional output. Liu *et al.* [114] used Lasso regularisation on channel saliencies to induce sparsity and prune channels with a global threshold. Ye *et al.* [185] learned to sparsify CNNs with an iterative shrinkage/thresholding algorithm applied to the scaling factors in batch normalisation. There are methods [116, 213] that use greedy algorithms for channel selection. Luo *et al.* [116] selectively prune input channels while minimizing the reconstruction error of the convolutional output. Zhuang *et al.* [213] similarly perform channel selection to minimise a joint loss of the reconstruction error and the classification power of convolutional channels. Huang *et al.* [78] and He *et al.* [71] adopted reinforcement learning to train agents to produce channel pruning decisions.

In general, existing work in the field of network pruning demonstrates that fine-grained pruning is better at achieving a greater reduction in terms of model sizes. However, when talking about real inference time reduction on commodity hardware such as GPUs, coarse-grained methods are a better choice.

2.3.2 Quantisation

Quantisation methods allow parameters to be represented with much narrower bit-widths than the 32-bit long floating-point numbers used in classic CPU- or GPU-based DNN implementations. Converting numbers to fixed-point representations drastically reduces computation and memory requirements [32, 109, 81]. An n-bit fixed-point number with a binary point position p can represent a value $\hat{x}$ with:

$$\hat{x} = 2^{-\mathsf{p}} \times m_{\mathsf{n}} m_{\mathsf{n}-1} \dots m_1, \tag{2.2}$$

Lin *et al.* claim that a suitable fixed-point quantisation can offer a large reduction in model sizes without any accuracy loss [109]. In extreme cases, the parameters can be binary (0 and 1) [80] or ternary values (-1, 0, 1) [104]. This significantly reduces the memory requirements for parameters, but naively putting weights and activation values to extreme low bit-widths can cause a huge decrease in network accuracy. The idea of grouping numbers with common bases might be long-existing [172], Lin *et al.* and Zhang *et al.* first showed that this technique works well with neural network model parameters [110, 192]. They group parameters using a set of shared numerical bases, so that the offsets of these bases can be quantised to binary or ternary values without a large decrease in network accuracy [110, 192]. These extreme low-precision number formats might be challenging to be exploited by general computing platforms but can bring a significant performance gain to custom hardware accelerators. Inference on general purpose hardware, *e.g.* CPUs and GPUs, often utilise 8-bit fixed-point numbers. The Intel DL Boost provides built-in instructions, an extension based on the Advanced Vector Extensions 512 (AVX-512), to accelerate 8-bit model inference [9]. Recently, Intel also released an Advanced Matrix extensions (AMX) in addition to AVX-512 [2]. Its supported tile matrix multiplication enables around 2000 INT8 operations per cycle per core. NVIDIA now has the TensorRT toolkit to support various low-precision formats (the lowest possible bit-width is 8-bit) on a variety of GPU devices [161].

Arithmetics beyond fixed-point have also being explored for efficient neural network inference. Miyashita *et al.* showed that logarithmic data representations can be effective on DNNs, and Lee *et al.* also discovered that powers-of-2 numbers is sufficient for networks to retain accuracy but replaces expensive multipliers with shifting [101]. Shift quantisation results in the following representable values, where $s \in \{-1, 0, 1\}$ indicates the sign of the value, b is a constant integer shared among weights within the same layer which ensures no values overflow, and e is a variable exponent:

$$\hat{x} = s \times 2^{e-\mathsf{b}}. \tag{2.3}$$

Many quantisation methods also consider re-training the quantised networks for a better post-quantisation accuracy, this is also known as quantisation-aware training (QAT). Courbariaux *et al.* first proposed that re-training the quantised networks can significantly reduce the accuracy loss introduced by quantisation errors [32]. Direct quantisation and quantisation-aware training are now pervasive in deep learning. Modern deep learning frameworks such as Pytorch [129] and Tensorflow [129] now both offer fixed-point quantisation and quantisation aware training to help developers to deploy deep learning models to mobile or more constrained devices.

2.3.3 Automated machine learning and network architecture search

Automated machine learning (AutoML) refers to the idea of automating the entire ML pipeline that involves four major steps: data preparation, feature engineering, model generation and model estimation [68]. While network architecture search (NAS) methods focus on automating two major processes of this whole AutoML pipeline, namely the model generation and model estimation phases.

The rise of NAS comes at a time when the manual design and tuning of architectures of DNNs on custom datasets is getting increasingly difficult. One challenge is the increase in the number of different possible operations that may be employed, *e.g.* in the field of computer vision, simple convolutions and fully connected layers [93] have expanded to include depth-wise separable convolutions [73], grouped convolutions [196], dilated convolutions [168], *etc.* This opens up a much larger design space for neural network architectures but makes manual architectural engineering a lot more challenging than before.

Network Architecture Search (NAS) seeks to automate the search for the best DNN architecture. A challenge in this search is how to properly define a search space of network architectures. One common practice is to represent neural architecture as a direct acyclic graph (DAG) consisting of ordered nodes. However, this DAG representation is potentially very large if we consider deep neural networks with hundreds of layers and each layer has a great number of possible operations. Several NAS methods [113, 130, 216, 206] then focused on a reduced cell-based search space, with the hope that learned cell structure can be repeated several times to compose a complete network. There is a clear motivation behind this cell-based search space: previous well-performing human-designed networks normally stack a number of fixed blocks [66]. Now if we can build a successful cell, it is expected that replicating this cell multiple times would construct a good model architecture. Figure 2.6 demonstrates a cell-based search space in DARTs [113]. Each red edge in $Cell_i$ is a searchable operation, the designer only searches for reduction and normal cells and repeat these cells several times to build a complete network. Another popular search space setup in NAS is to construct a supernet from a seed/backbone network [17, 18, 157]. The backbone network normally relies on a pre-built successful network such as ResNets [66] or MobileNets [73]. The NAS methods then expand this network with a set of pre-defined candidate operations to build a supernet. Finally, the NAS method will exploit a search method (*e.g.* genetic algorithms) to pick the best network (architecture adaption phase in Figure 2.7). It is generally hard to argue which search space is better, since they have different limitations. Cell-based architecture relies on repeating the blocks and its complex DAG of each cell might induce a large inference latency (I discuss this latency problem in details in Chapter 5). The supernet-based approach, on the other hand, focuses on its pre-defined backbone and can have vastly different performances when backbones are not designed with care.

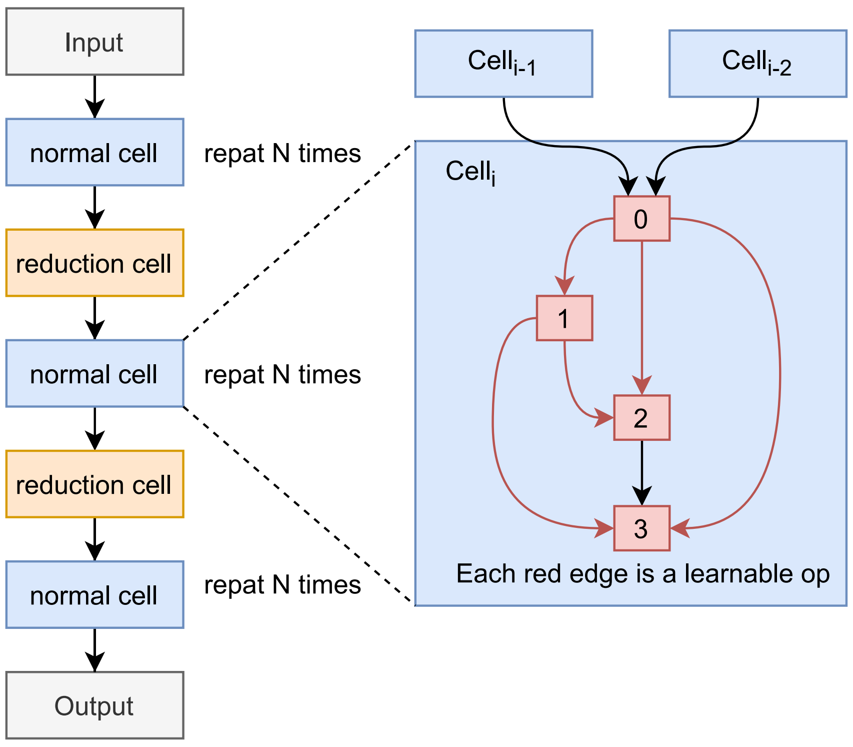


Figure 2.6: An illustration of a cell-based NAS search spaces.

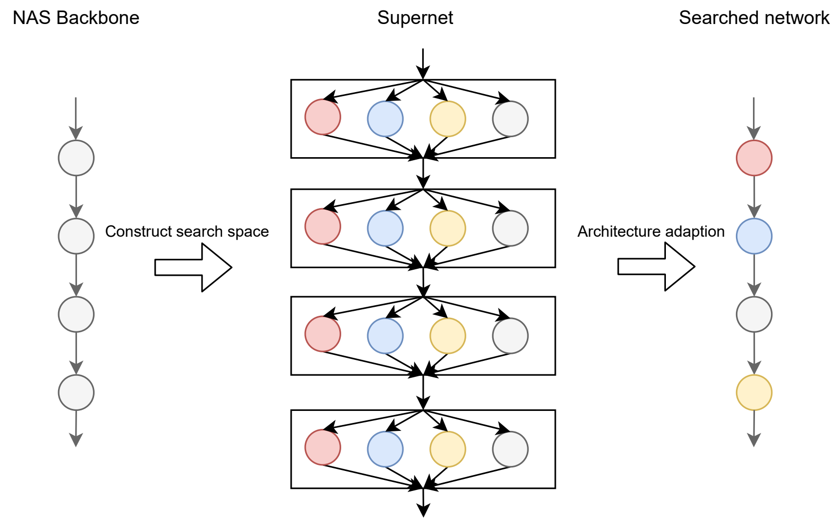


Figure 2.7: An illustration of a supernet-based NAS search spaces.

NAS methods can also be classified by their search methodologies. Early work in this field, such as NASNet [216] and AmoebaNet [136], employed reinforcement-learning (RL) or evolutionary algorithms (EA) to discover optimal network architectures, but this comes at a large computational cost. Each update of the controller requires a few hours to train a child network to convergence which significantly increases the search time. Liu *et al.* proposed Differentiable Architecture Search (DARTS) that is a purely gradient-based optimisation (GO); each candidate operation's importance is scored using a trainable scalar and updated using Stochastic Gradient Descent (SGD) [113]. Subsequently, Casale *et al.* approached the NAS problem from a probabilistic view, transforming concrete trainable scalars used by DARTS [113] to probabilistic priors and only train a few architectures sampled from these priors at each training iteration [20]. Wu *et al.* and Xie *et al.* used the Gumbel-softmax trick to relax discrete operation selection to continuous random variables [175, 177]. I present the representative NAS techniques in Table 2.1. It can be concluded that these three mainstream NAS approaches all demonstrate compatible search time in terms of GPU days.

There is also now a rise of applying NAS with hardware design parameters, so that both the network architecture and hardware architecture search spaces are jointly explored. Abdelfattah *et al.* and Zhou *et al.* have explored how NAS can establish an efficient exploration in the software-hardware co-design search space [4, 211].

2.4 Neural network hardware accelerators

In the last few years, a great number of custom hardware accelerators have been proposed and implemented for improving the run-time efficiency of DNNs. In this section, I would

Method	Style	Top1/Top5 (%)	Params (Millions)	GPU Days
NASNet [216]	RL	82.7/96.2	88.9	2000
Path-level EAS [16]	RL	74.6/91.9	594	200
FPNAS [33]	RL	73.3/-	3.41	0.8
AmoebaNet [136]	EA	82.8/96.1	86.7	3150
OFA [18]	EA	80.0/-	-	1.7
SMASH [14]	GO	61.4/83.7	16.2	3
PARSEC [20]	GO	74.0/91.6	5.6	1
ProxylessNAS [17]	GO	75.1/92.5	-	8.33
SNAS [177]	GO	72.7/90.8	4.3	1.5
FBNet [175]	GO	74.9/-	5.5	9
DARTS [113]	GO	73.3/91.3	4.7	4

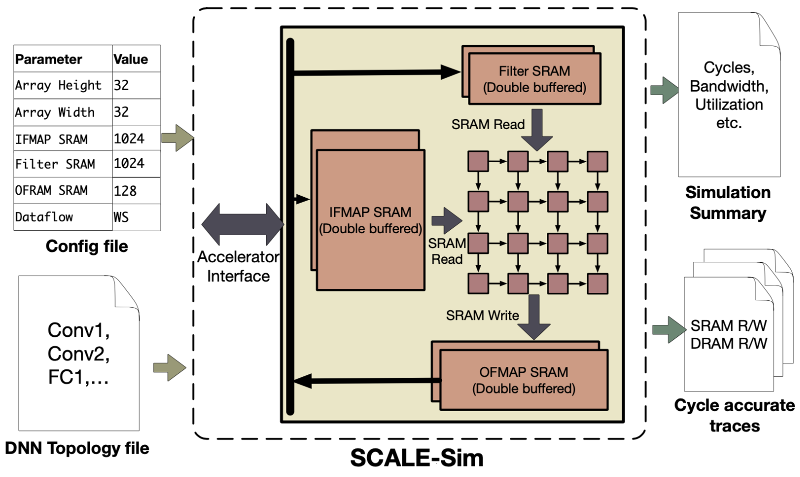
Table 2.1: Different Network Architecture Search Methods and their performance on the ImageNet 2012 dataset [93]. RL, EA and GO represent Reinforcement Learning, Evolutionary Algorithm and Gradient-based Optimisation respectively.

like to compare some popular accelerator designs in terms of their processing elements and data reuse models. Energy consumed by a hardware accelerator can be broadly categorised as relating to computation, on-chip memory accesses or off-chip memory accesses. The design of DNN accelerators need to carefully arrange its available hardware resources to processing elements, on-chip memory systems to improve efficiency.

2.4.1 From general-purpose processors to custom hardware accelerators

General-purpose processors are the most common hardware in the world. These processor, particularly Central Processing Units (CPUs), are designed to be efficient for general-purpose computing. However, these computing platforms are not friendly to massively parallel workloads such as DNN inference. For operations in DNN inference, many of these matrix multiplications can be mapped efficiently to a single-instruction, multiple=thread (SIMT) or single-instruction, multiple data (SIMD) computation paradigm. This is also the reason for the dominant usage of GPUs as the computation platform for DNN inference. However, there is also a rising trend of using custom computing to accelerate DNN inference. Hardware platforms such as Field Programmable Gate Arrays (FPGA) and Application-Specific Integrated Circuits (ASICs) are designed to be efficient for DNN inference. While GPUs excel in computing dense float-point matrix multiplies, these platforms are not designed for short bit-width fixed-point operations and sparse computation. Researchers have then looked at how customising the computation units together with the memory system can be used to achieve better performance, and we will review some of these hardware designs in the rest of this chapter.

One particularly popular hardware architecture for DNN inference is the Systolic Array. This is a homogeneous network of processing elements that are connected to a single memory system. These processing elements are normally data processing units (DPUs), and each of these DPUs can be configured to independently compute a partial result as the data moves from its upstream neighbors. The DPU then performs the computation and then passes the result to its downstream neighbors, as illustrated in Figure 2.8.



Parameter	Value
Array Height	32
Array Width	32
IFMAP SRAM	1024
Filter SRAM	1024
OFRAM SRAM	128
Dataflow	WS

Figure 2.8: An illustration of a systolic-array based accelerator presented in the ScaleSIM simulator [139]. The processing elements are shown as small colored blocks in the middle.

2.4.2 Processing elements

The design of processing elements are heavily impacted by the network compression algorithms used. The complexity of a PE's compute logic is highly dependent on data format (arithmetic) and bit-width, this in turn, also has implications for area, power efficiency and latency. One consideration is whether the processing elements are zero-aware, meaning that they can take advantage of the sparsities in both weights and feature maps. Bit-widths can heavily impact the arithmetic cores in a neural network and thus the computation efficiency and latency.

In this sub-section, I only review representative compression-aware DNN accelerators with unique processing elements. EIE encodes weights into 4-bit to reduce the bandwidth requirements [63]. This, in turn, requires the EIE accelerator to have decoding units in hardware. In addition, EIE supports non-zero detection, so that the run-time sparsities in activations can be exploited. This non-zero zero detection requires an activation queue to store non-zero activations. Similarly, Eyeriss supports sparse activations as well: it misses both the memory read and calculation if detected a zero value on activations. Angle-Eye [56] is a standard accelerator with dense weights and activations. Nullhop [5]

Accelerator	Sparsity	Bit-widths
Diannao [24]	Dense weights and dense activations	16-bit fixed-point PE
EIE [63]	Sparse weights and sparse activations	4-bit encoding, 16-bit fixed-point PE
SCNN [128]	Sparse weights and sparse activations	4-bit encoding, 16-bit fixed-point PE
Eyeriss [27]	Dense weights and sparse activations	16-bit fixed-point PE
Angel-Eye [56]	Dense weights and dense activations	16-bit or 8-bit fixed-point PE
Nullhop [5]	Dense weights and sparse activations	16-bit fixed-point PE
XNOR Neural Engine [31]	Dense weights and dense activations	binary PE with partial sums stored as 16-bit
Bit-Tactical [35]	Bit-level sparse weights and activations	16-bit fixed-point PE
YodaNN [7]	Dense weights and dense activations	binary weights, 12-bit activations

Table 2.2: Sparsities and data formats of various neural network accelerators.

exploits activation sparsity to reduce the computation complexity. XNOR Neural Enginee focuses on accelerating binary neural networks [31], and Bit-Tactical [35] accelerates neural networks by exploiting bit-level sparsities using bit-serial operators. YodaNN is an accelerator for binarised neural networks, the arithmetic operations are nearly costless because they are simply bit-wise operations.

2.4.3 Data reuse models

Consider the nested for loops in Listing 2.1 again, we can identify these iterators: [i, j, fi, fo, col, row]. Table 2.3 shows how recently proposed hardware accelerators unroll different loop iterators.

Accelerator	Unrolled iterators
Eyeriss [27]	i, j, row, col
EIE [63]	i, j and fo
Ma *et al.* [117]	fi and fo
Rahman *et al.* [134]	col, row and fo

Table 2.3: Unrolling strategies of different neural network accelerators.

Unrolling different iterators in hardware would have different impacts on the underlying architecture (PE arrangements, interconnects and etc) and data reuse model. Naive unrolling of only i, j means exploiting the hardware parallelism inside each kernel, however, these kernels are small in size and thus does not provide enough hardware parallelism. Eyeriss tries to obtain more parallelisms by unrolling the row and col iterators together with the kernels (i, j) [27]: it utilizes a systolic-array like architecture where outputs are streamed vertically to be summed and inputs are streamed diagonally. The weights are broadcasted to each processing element using a large multicast network. Ma *et al.* find the parallelisms in input feature maps (fi): each input feature map is fully calculated before switching to a new one [117]. The partial sums for output feature maps (fo) are cached on-chip until the calculation finishes. In this case, only weights are not reused. In contrast, Rahman *et al.* choose to fully unroll the output feature maps and their corresponding dimensions (row, col, fo) in hardware [134], however, these unrollings force partial sums at the output to be store back into the external memories if on-chip storage is not large enough.

In short, various unrollings indicate different hardware architectures (PE design, on-chip interconnects, memory accesses, *etc.*) and data reuse models. Convolution layers normally vary in terms of channel count, kernel size and feature map sizes. An optimal hardware for one particular convolution layer does not apply to another: hardware that is optimised for one particular convolution layer may perform poorly when used to execute different layers with different dimensions.

2.4.4 A comparison of DNN hardware accelerators

Table 2.4 summarises key data for a representative sample of recent hardware platforms used to execute DNNs.. NVIDIA GPUs are included as a baseline platform: the 1080Ti GPU is a server level chip and TK1 aims for embedded systems.

Hardware	Power	Throughput	Technology	Clock Frequency	Bit-width
Edge/Embedded Devices					
NVIDIA TK1	10.2W	155 GOPS	28nm	852 MHZ	32b Float
Eyeriss [27]	0.28W	84 GOPS	65nm	200 MHz	16b Fixed
EIE [63]	0.59W	102 GOPS	45nm	800 MHz	16b Fixed
ShiDianNao [41]	0.32W	194 GOPS	65nm	1 GHz	16b Fixed
AngleEye [56]	3.5W	188 GOPS	28nm (FPGAs)	150 MHZ	8b Fixed
Cloud Devices					
NVIDIA 1080Ti	250W	10.6 TOPS	16nm	1580 MHz	32b Float
Brainwave [29]	225W	90.0 TOPS	14nm (FPGAs)	500 MHz	8b MS-Float
GraphCore MK1 IPU [84]	250W	124.5 TOPS	16nm	1.6 GHz	32b/16b Float mixed

Table 2.4: Performance comparison of neural network accelerators. Bit-width means the bit-width of the computation cores.

AngleEye and Brainwave are two representative FPGA-based accelerator designs [56, 29]. As shown in the table, for FPGA implementations I choose to report performance assuming 8-bit processing elements, which shows the flexibility of using reconfigurable hardwares. The Brainwave accelerator utilises a special data format that is similar to a block floating-point arithmetic [87]. In general, FPGA based accelerators tend to use special or lower precision arithmetics because of the fine-grained reconfigurability, ASIC based accelerators have better energy efficiencies mainly because of the higher clock frequency.

In Figure 2.9, I consider a wider range of cores from both industry and academia. The vertical axis shows the number of operations per unit die size (mm^2) and the horizontal axis is the years that these accelerators are published. I calculate the number of operations

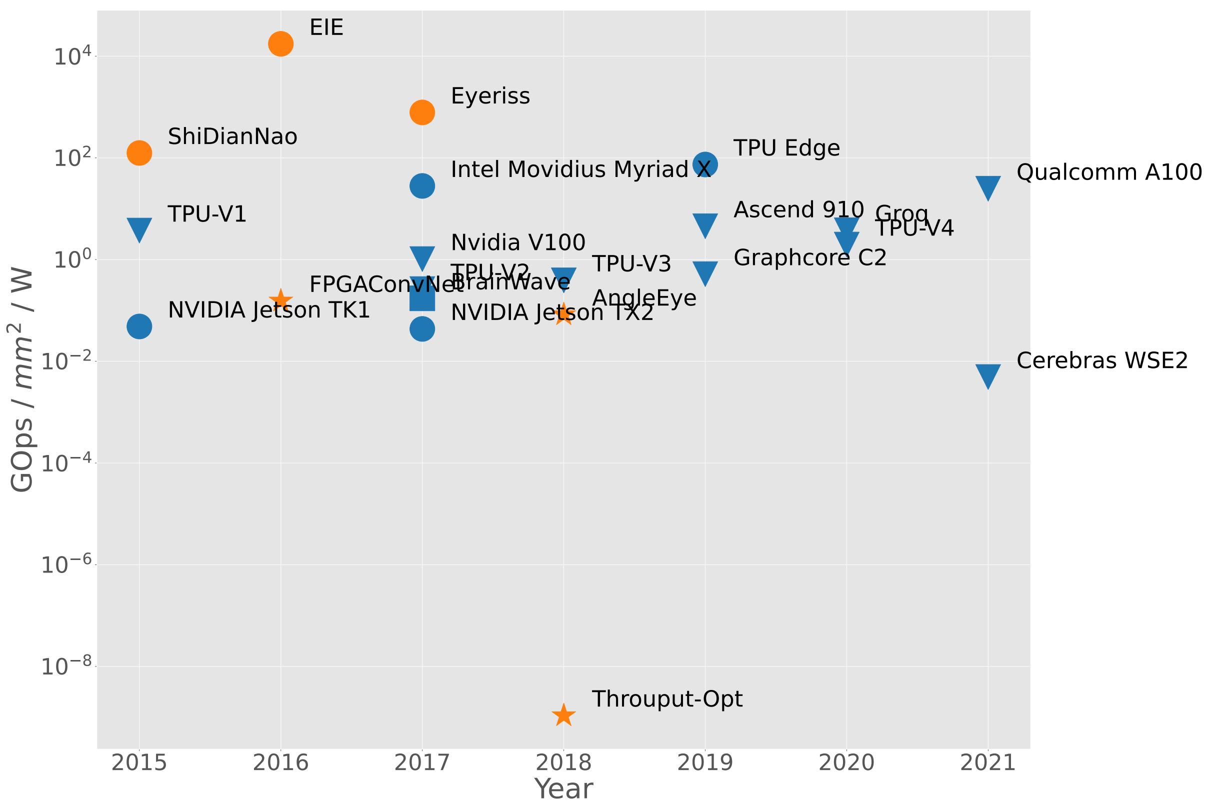


Figure 2.9: A comparison of various DNN accelerators. Orange circles represent research ASIC accelerators for embedded systems, blue circles represent commercial ASIC accelerators for embedded systems, blue triangles are commercial ASIC cores for cloud systems, blue rectangles are commercial FPGA designs for cloud systems and orange starts are research FPGA designs for embedded systems.

from the fastest arithmetic that the accelerator can offer. For instance, if an accelerator supports both INT16 and FLOAT16, I report the number of operations as the number of INT16 operations finished per unit time. The estimation is coarse-grained for several platforms because of the limited information available, *e.g.* Cerebras does not report the exact number of floating-point operations so I have to estimate it from its available clock frequency and number of processing elements. In particular, the following hardware systems are considered:

- ASIC accelerators for embedded systems (orange circles and blue circles): ShiDianNao [41], EIE [63] and Eyeriss [27] are representative designs from the academia. These accelerators tend to use a more aggressive optimisation strategy on the software space (*e.g.* more compact number formats). In particular, EIE utilised fine-grained pruning, fixed-point quantisation and Huffman encoding to aggressively compress the model [63]. Nvidia Jetson [159, 120], Intel Movidius [3] and TPU Edge [21] have similar performance although being designed at different years, and I consider them as representative designs from the industry.

- FPGA-based accelerators for embedded systems (orange stars): FPGAConvNet [164] and AngleEye [56] are two representative designs from the academia.
- ASIC accelerators for cloud systems (blue triangles): TPU-V1, TPU-V2, TPU-V3 and TPU-V4 are a series of tensor processing units developed at Google [85]. TPU-V1 and V4 supports network inference only whereas the other two generations support both training and inference. Nvidia V100 is a popular platform in popular cloud services [118]. Huawei HiSilicon ships the Ascend 910 chip based on the same Huawei Da Vinci architecture [108]. The Groq Tensor Streaming Processor are a set of Superlanes which each Superlane contains a specialised very long instruction word (VLIW) instruction [60]. Graphcore AI released a C2 card, showing strong inference performance for both ResNet50 training and inference [86]. Cerebras shows an incredibly large chip using their waver scaling technology on the 7nm technology node [1].
- FPGA-based accelerators for cloud systems (blue triangles): In this case, BrainWave from Microsoft might be the most recognised FPGA design that is widely deployed in today' Microsoft data centers [48].

In Figure 2.9, I observe several interesting trends that is worth highlighting in this dissertation:

- DNN inference accelerators for edge systems designed by researchers (ShiDianNao, EIE, Eyeriss) show the best performances, they outperform other accelerators by a significant margin. These accelerators tend to use aggressive optimisation techniques. For instance, EIE [63] utilised pruning, quantisation and encoding. FPGA accelerators show the same trend, where research accelerators show better efficiency compared to commercial grade accelerators.
- The middle band in Figure 2.9 (blue circles) are mainly embedded systems designed by the industry. These accelerators show a better performance compared to server-class accelerators. This indicates that simply using a larger die size (making larger chips) might increase the absolute operations per second counts but does not provide a better power efficiency per die size. This phenomenon is easy to understand since larger die sizes normally suggest an increased number of transistors and thus a more difficult routing. In addition, server-class chips normally integrate more networking facilities and DRAM channels. On the other hand, loading off-chip data from DRAM is likely to be a bottleneck for server-class accelerators.
- DNN accelerators designed by the industry do not show great performance (Gops/mm^2/W) in Figure 2.9. It is commonly known that industry accelerators will have to support

more types of neural networks and even ensure backwards compatibility [85], this wider coverage forces the designers to use more conservative optimisation methods. Some of these industry accelerators have particular applications to support with particular quality targets, *e.g.* in some cases forcing the use of some forms of floating-point arithmetics. They may also have to take and run networks as they are, i.e. they don't have the ability to retrain after applying some sort of compression technique.

It is also worth mentioning that this comparison does not consider the effect of different fabrication processes. However, it is generally expected that, at least in industry, the accelerators fabricated in later years are using a more advanced nanometer technology. On the other hand, recent DNN accelerators contain both ASIC and FPGA designs. ASIC designs have a major advantage of its high clock frequency FPGAs used to be a testing facility for ASIC designs [95], but now start to rise as a general purpose computation platform because of the following reasons:

- Faster time to market: ASIC designs has a much longer development cycle compared to FPGAs.
- Design difficulty: The design process of ASIC accelerators is complex and requires collaborations of different entities, FPGAs, on the other hand, has a much simpler design flow, especially with the rise of recent high-level-synthesis tools [173].
- Flexibility: ASIC accelerators have to focus on a set of pre-defined applications, PGA is a flexible device due to its hardware re-programmability.

In this dissertation, I mainly focus on FPGA designs and provide a more thorough discussion of DNN accelerating on FPGAs in Chapter 4.

Chapter 3

Pruning and quantisation for efficient neural networks

In this chapter, I will look at how pruning and quantization techniques can reduce the amount of computation required for neural network inference. These techniques are often used in the context of deep learning with a pre-defined DNN architecture and pre-trained DNN weights, and the objective is to lower the amount of computation required for a neural network is often a limiting factor.

I first discuss the intriguing properties of deep neural networks: their redundancy and ability to recover accuracy from re-training. I start with a discussion of the effects of combining different basic neural network compression methods and a flexible software tool for achieving this combined compression. I later demonstrate and evaluate two novel compression techniques, dynamic coarse-grained pruning and focused quantisation. Finally, I discuss how various approximate computing methods might be applicable to different hardware platforms.

This chapter includes relevant contents published in:

- Yiren Zhao, Xitong Gao, Robert Mullins, and Chengzhong Xu. Mayo: A framework for auto-generating hardware friendly deep neural networks. In *Proceedings of the 2ndInternational Workshop on Embedded and Mobile Deep Learning, pages 25–30, 2018. (EMDL 2018)*

- Xitong Gao, Yiren Zhao, Lukasz Dudziak, Robert Mullins, Chengzhong Xu. Dynamic channel pruning: Feature boosting and suppression. In *International Conference of Learning Representations 2019 (ICLR 2019).*

- Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. Focused quantisation for sparse CNNs. In *Advances in Neural Information Processing Systems,pages 5584–5593, 2019 (NeurIPS 2019)*

Yiren Zhao proposed the idea of constructing the Mayo framework. Xitong Gao developed the pruning functions and Yiren Zhao constructed the quantisation functions for Mayo. Yiren Zhao conceived and designed the experiments for Dynamic channel pruning and Focused quantisation. Xitong Gao enhanced the implementation of Dynamic channel pruning with normalisation and achieved better results, he also provided insights on developing the feature saliency motivation in the publication.

3.1 Intriguing properties of DNNs

DNNs are, in general, inherently redundant, which means that many operations in DNNs compute highly-correlated results. Wen *et al.* showed that filters (also referred as kernels in some literature) of a network have a similar low-rank basis if decomposed using singular value decomposition [170]. From the learning point of view, Srivastava *et al.* proposed Dropout that simply removes neurons in a DNN while training and it improves the training quality and prevents overfitting [153]. Since many neurons tend to learn similar representations, stochastically dropping neurons helps to enhance the capabilities of singular neurons. In other words, Dropout works by breaking the neuron-wise correlations, which in turn proves that many neurons are computing highly correlated results.

Another interesting property of neural networks is their ability to recover accuracy from further training. Han *et al.* proposed Deep Compression that achieves a $35\times$ compression rate using the AlexNet [93] model with re-training at every compression step. The Deep Compression flow includes pruning, quantisation and Huffman encoding. The pruning process suppresses a significant part of the weights to zero, and quantisation employs a standard low-precision fixed-point arithmetic. At the end, the encoding scheme uses a compact representation for the quantised sparse tensors to offer a further reduction. At each compression stage in Deep Compression, the authors conduct re-training. Re-training allows learnable parameters in a network to adapt to the damages incurred during compression.

These intriguing properties of DNNs have driven research that focuses on iteratively trimming and re-training models to reduce the runtime computational requirements.

3.2 Combining different compression techniques

As illustrated in Chapter 2, a wide range of compression techniques have proven to be effective for lowering the computation and memory requirements of DNNs. Pruning directly reduces the number of connections and quantisation methods enable each parameter to be represented with much narrower bit-width than the 32-bit single-precision floating-point numbers.

Han's Deep Compression further demonstrated the possibility of achieving multiplying gains in terms of shrinking the model sizes when applying a series of compression techniques [64]. Deep Compression only investigates a single combination of pruning, quantisation and encoding techniques in the large compression optimisation space. Intuitively, there are potentially many other combinations of compression techniques that have equivalent accuracy but with very different hardware implications. In general, we will be presented with the possibility of combining many different compression techniques and selecting from numerous different number formats. The Mayo tool was designed to help explore this design space and find optimal solutions. Mayo then aims to be a flexible software compression optimisation framework to support arbitrary combinations of various compression techniques on different components of a neural network as illustrated in Figure 3.1.

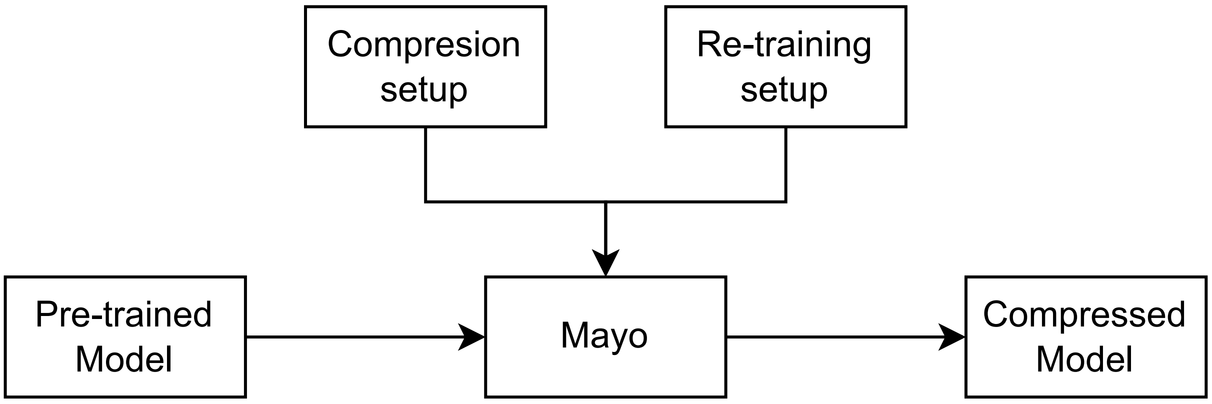


Figure 3.1: An high-level illustration of the Mayo tool. Mayo takes a set of compression techniques and combines them to achieve a single compression rate. The compression setup and the re-training setup can be configured in YAML files. The tool takes a pre-trained model as an input and outputs a compressed model.

At the time of implementing Mayo, the research community had a few open source methods and frameworks for compressing neural networks. Table 3.1 demonstrates the differences between Mayo and other existing tools, it is clear that Mayo supports a wider range of pruning and quantisation methods with the ability to apply these methods on different components of a neural network. The dynamic fixed-point arithmetic refers to a specialised fixed-point arithmetic where the whole layer shared a scaling factor for all the fixed-point numbers of that layer.

All compression methods are implemented as objects called *overriders* in Mayo. Overriders can be flexibly applied to not only layer-wise model parameters $\boldsymbol{\theta}_l$, but also the underlying algorithm f_l, and even the gradient of each layer computation $\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \boldsymbol{\theta}_l)$ where $\mathbf{x}_l \in \mathbb{R}^{C_l \times H_l \times W_l}$,, The design of overriders provides an abstraction for various compression techniques.

An overrider $g \in G$, configured with suitable hyperparameters, takes a multi-dimensional array (a tensor) as an input, and produces a new array with the same shape as the

		Mayo	Ristretto	ADaPTION	DoReFa
Pruning	fine-grained	✓	✗	✗	✗
	coarse-grained	✓	✗	✗	✗
Quantization	fixed-point	✓	✓	✓	✓
	dynamic fixed-point	✓	✓	✓	✗
	mini-float	✓	✓	✓	✗
	log	✓	✗	✗	✗
	shift	✓	✓	✓	✗
Layer-wise customization		✓	✓	✓	✗
Automated hyperparameter optimization		✓	✗	✗	✗
Compression method chaining		✓	✗	✗	✗
Customizable components		w/a/g	w/a	w/a	w/a/g
Configuration format		YAML	Caffe	Caffe	Python

Table 3.1: A comparison: Ristretto [61], ADaPTION [119], DoReFa [210] and Mayo.

compressed variant. Parameters $\boldsymbol{\theta}_l$ can thus be simply substituted using any overrider $g_{\text{param}} \in G$:

$$\tilde{\boldsymbol{\theta}}_l = g_{\text{param}}(\boldsymbol{\theta}_l). \tag{3.1}$$

Moreover, overriders can be used to customise other trainable/non-trainable components of a DNN. For example, I can customise the activation function to replace f_i with $\tilde{f}_i$, using an overrider $g_{\text{activation}} \in G$, where for any input $\mathbf{x}_{i-1}$ and parameters $\boldsymbol{\theta}_l$, I have:

$$\tilde{f}_i(\mathbf{x}_{i-1}, \boldsymbol{\theta}_l) = g_{\text{activation}}\left(f_i\left(\mathbf{x}_{i-1}, \boldsymbol{\theta}_l\right)\right). \tag{3.2}$$

Under the same principle, overriders can be applied on tensor gradients.

Finally, overriders are recursively-compositional. Multiple overriders can be chained in sequence, which in turn provides greater compression opportunities. For example, given a pruning overrider g and a quantising overrider h, the composition of them is also an overrider that can be applied to any components, *i.e.* $h \circ g \in G$. In this case, for any parameters $\boldsymbol{\theta}_l$:

$$\tilde{\boldsymbol{\theta}}_l = h\left(g\left(\boldsymbol{\theta}_l\right)\right). \tag{3.3}$$

I show in Table 3.2 the effect of applying quantisation on top of a sparse CifarNet (details of this network is explained in later section in Table 3.3) [201] for classifying CIFAR10 [94] objects. I designed the CifarNet model based on the VGG family [151] but added Batch Normalizations and Dropout layers. Applying both quantisation and pruning offers a multiplying performance gain in terms of the compression rates. Table 3.2 shows that a pruned model, when quantised with dynamic fixed-point numbers, brings an over

Method	Bit-width	Density	Compression rate	Top-1/top-5 accuracy
baseline	32	100.00%	-	91.37%/99.67%
fixed-point (fixed)	4	100.00%	8.00×	89.64%/99.74%
dynamic fixed-point (DFP)	4	100.00%	8.00×	90.63%/99.68%
fine-grained pruning (pruned)	32	15.65%	6.39×	91.12%/99.70%
pruned + fixed	6	15.65%	33.92×	90.59%/99.68%
pruned + DFP	6	15.65%	33.92×	91.04%/99.70%

Table 3.2: Quantisations on sparse and dense CifarNets [201].

30× decrease in model sizes with a minimal decrease in classification accuracy (less than 0.4%).

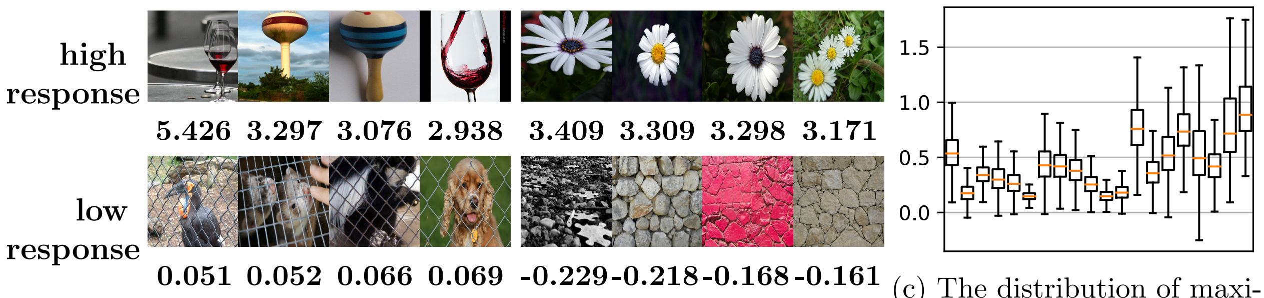
The comparison in Table 3.2 provides an interesting insight: *pruned + DFP* outperforms the *pruned + fixed* flow utilised in Deep Compression, this proves that there are further compression opportunities if a more flexible combination of pruning and quantisation strategy is permitted. *pruned + fixed* and *pruned + DFP* have the same compression rates but different accuracy values. This further demonstrates that there are many combinations of compression techniques that offer sub-optimal performance on accuracy, the developed Mayo framework is a suitable tool for traversing the design space. In addition the Mayo framework quantises not only the weights but also activation values, the tool also supports compressing each layer of the network independently, *i.e.* using different compression techniques or parameters. I will revisit this ability in Chapter 4 to demonstrate how these features can benefit the design of hardware accelerators.

3.3 Dynamic coarse-grained pruning

3.3.1 Static and dynamic pruning

One common approach to reduce the memory, bandwidth and computation costs is to prune over-parameterized CNNs, as discussed in Section 3.2. If performed in a coarse-grain manner this approach is known as *coarse-grained pruning* or *channel pruning* [185, 71]. One well-established approach in this domain is *Network Slimming* [114], this method considers the Lasso regularization on channel saliencies. The authors then introduce a scaling factor for each channel, which is multiplied to the output of that channel. Then *Network Slimming* jointly trains the network weights and these scaling factors, with the Lasso sparsity regularization imposed on the latter. Finally *Network Slimming* prunes those channels with small factors, and fine-tunes the pruned network.

Most channel pruning methods follow the same approach, they evaluate channel saliency and remove all input and output connections from unimportant channels, generating a smaller dense model. A saliency-based pruning method utilises static channel information, however, has two disadvantages. Firstly, by removing channels, the capabilities of CNNs are permanently lost, and the resulting CNN may never regain its accuracy for difficult inputs for which the removed channels were responsible. Secondly, the saliency of a neuron is not static, which can be illustrated by the feature visualization in Figure 3.2a, Figure 3.2b and Figure 3.2c. Here, a CNN is shown a set of input images, certain channel neurons in a convolutional output may get highly excited, whereas another set of images elicit little response from the same channels. This is in line with my understanding of CNNs whereby neurons in a convolutional layer specialise in recognizing distinct features, and the relative importance of a neuron depends heavily on the inputs.



(a) Channel 114 (b) Channel 181 (c) The distribution of maximum activations

Figure 3.2: When images from the ImageNet validation dataset are shown to a pre-trained ResNet-18 [66], the outputs from certain channel neurons may vary drastically. The top rows in (a) and (b) are found respectively to greatly excite neurons in channels 114 and 181 of layer `block_3b/conv2`, whereas the bottom images elicit little activation from the same channel neurons. The number below each image indicate the maximum values observed in the channel before adding the shortcut and activation. Finally, (c) shows the distribution of maximum activations observed in the first 20 channels.

The above shortcomings prompt the question: why should we prune by static importance, if the importance is highly input-dependent? Surely, a more promising alternative is to prune dynamically depending on the current input. A dynamic channel pruning strategy allows the network to learn to prioritise certain convolutional channels and ignore irrelevant ones. Instead of simply reducing model size at the cost of accuracy with pruning, I can accelerate convolution by selectively computing only a subset of channels predicted to be important at run-time, while considering the sparse input from the preceding convolution layer. In practice, the amount of activation values stayed on-chip and the number of DRAM reads, writes and arithmetic operations used by a well-designed dynamic model can be almost identical to an equivalently sparse statically pruned one. In addition to saving computational resources, a dynamic model preserves all neurons of the full model, which minimises the impact on task accuracy.

3.3.2 Feature boosting and suppression

Motivated by the shortcomings described above, I will start with a high-level illustration (Figure 3.3) of how *Feature Boosting and Suppression* accelerates a convolutional layer with *batch normalization* (BN). The auxiliary components (in red) predict the importance of each output channel based on the input features, and amplify the output features accordingly. Moreover, certain output channels are predicted to be entirely suppressed (or zero-valued as represented by ▢), such output sparsity information can advise the convolution operation to skip the computation of these channels, as indicated by the dashed arrow. It should be noted that convolutions can be skipped due to (1) inactive input channels and (2) due to the suppression of particular output channels. The rest of this section provides a detailed explanation of the components in Figure 3.3.

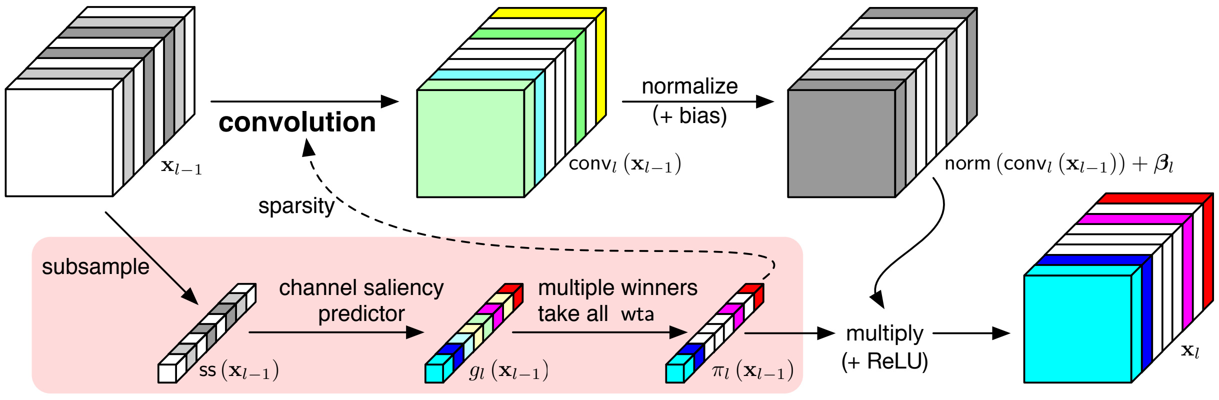


Figure 3.3: A high level view of a convolutional layer with FBS. By way of illustration, I use the l^{th} layer with 8-channel input and output features, where channels are colored to indicate different saliencies, and the white blocks (▢) represent all-zero channels.

Mathematically, consider the following generalization of a layer with dynamic execution:

$$\hat{f}(\mathbf{x}, \cdots) = f(\mathbf{x}, \boldsymbol{\theta}, \cdots) \cdot \pi(\mathbf{x}, \boldsymbol{\phi}, \cdots), \tag{3.4}$$

where f and π respectively use weight parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ and may have additional inputs, and compute tensors of the same output shape, denoted by $\mathbf{F}$ and $\mathbf{G}$. Intuitively, the expensive $\mathbf{F}^{[\mathbf{i}]}$ can always be skipped for any index $\mathbf{i}$ whenever the cost-effective $\mathbf{G}^{[\mathbf{i}]}$ evaluates to $\mathbf{0}$. Here, the superscript $[\mathbf{i}]$ is used to index the $\mathbf{i}^{\text{th}}$ slice of the tensor. For example, if we have features $\mathbf{F} \in \mathbb{R}^{C \times H \times W}$ containing C channels of H-by-W features, $\mathbf{F}^{[c]} \in \mathbb{R}^{H \times W}$ retrieves the c^{th} feature image. I can further sparsify and accelerate the layer by adding, for instance, a Lasso on π to the total loss, where $\mathbb{E}_{\mathbf{x}}[\mathbf{z}]$ is the expectation of $\mathbf{z}$ over $\mathbf{x}$:

$$\mathcal{R}(\mathbf{x}) = \mathbb{E}_{\mathbf{x}}\left[\|\pi(\mathbf{x}, \boldsymbol{\phi}, \cdots)\|_1\right], \tag{3.5}$$

Despite the simplicity of this formulation, it is however very tricky to design $\hat{f}$ properly. Under the right conditions, I can arbitrarily minimise the Lasso while maintaining the same output from the layer by scaling parameters. For example, in low-cost collaborative layers [39], f and π are simply convolutions (with or without ReLU activation) that respectively have weights $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Since f and π are homogeneous functions, one can always halve $\boldsymbol{\phi}$ and double $\boldsymbol{\theta}$ to decrease Equation (3.5) while the network output remains the same. In other words, the optimal network must have $\|\boldsymbol{\phi}\|_\infty \to 0$, which is infeasible in finite-precision arithmetic. For the above reasons, Dong *et al.* [39] observed that the additional loss in Equation (3.5) always degrades the CNN's task performance. Ye *et al.* [185] pointed out that gradient-based training algorithms are highly inefficient in exploring such reparameterisation patterns, and channel pruning methods may experience similar difficulties. Shazzer *et al.* [145] avoided this limitation by finishing π with a softmax normalization, but the Equation (3.5) can no longer be used as the softmax renders the ℓ^1-norm, which now evaluates to 1. In addition, similar to sigmoid, softmax (without the cross entropy) is easily saturated, and thus may equally suffer from the gradient vanishing problem.

Instead of imposing sparsity on features or convolutional weight parameters [169, 6, 105, 69], channel pruning methods [114, 185] induce sparsity on the BN scaling factors $\boldsymbol{\gamma}_l$. Inspired by them, FBS similarly generates a channel-wise importance measure. Yet contrary to them, instead of using the constant BN scaling factors $\boldsymbol{\gamma}_l$, FBS predicts channel importance and dynamically amplify or suppress channels with a parametric function $\pi(\mathbf{x}_{l-1})$ dependent on the output from the previous layer $\mathbf{x}_{l-1}$. Here, I propose to replace layer definition $f_l\left(\mathbf{x}_{l-1}\right)$ for all $l \in [1, L]$ with $\hat{f}_l\left(\mathbf{x}_{l-1}\right)$ which employs dynamic channel pruning:

$$\hat{f}_l\left(\mathbf{x}_{l-1}\right) = \left(\pi_l\left(\mathbf{x}_{l-1}\right) \cdot \left(\mathsf{norm}\left(\mathsf{conv}_l\left(\mathbf{x}_{l-1}, \boldsymbol{\theta}_l\right)\right) + \boldsymbol{\beta}_l\right)\right)_+ , \tag{3.6}$$

where norm is batch normalisation and a low-overhead policy $\pi_l\left(\mathbf{x}_{l-1}\right)$ evaluates the pruning decisions for the computationally demanding $\mathsf{conv}\left(\mathbf{x}_{l-1}, \boldsymbol{\theta}_l\right)$:

$$\pi_l\left(\mathbf{x}_{l-1}\right) = \mathsf{wta}_{\lceil dC_l \rceil}\left(g_l\left(\mathbf{x}_{l-1}\right)\right). \tag{3.7}$$

Here, $\mathsf{wta}_k(\mathbf{z})$ is a k-winners-take-all function, *i.e.* it returns a tensor identical to $\mathbf{z}$, except that I zero out entries in $\mathbf{z}$ that are smaller than the k largest entries in absolute magnitude. In other words, $\mathsf{wta}_{\lceil dC_l \rceil}(g_l(\mathbf{x}_{l-1}))$ provides a pruning strategy that computes only $\lceil dC_l \rceil$ most salient channels predicted by $g_l(\mathbf{x}_{l-1})$, and suppresses the remaining channels with zeros. In addition, $g_l(\mathbf{x}_{l-1})$ is a simple fully connected layer that learns to predict channel saliencies using only a handful of learning parameters.

It is notable that the proposed strategy prunes $C_l - \lceil dC_l \rceil$ least salient output channels from l^{th} layer, where the density $d \in]0, 1]$ can be varied to sweep the trade-off relationship between performance and accuracy. Moreover, pruned channels contain all-zero values. This allows the subsequent $(l+1)^{\text{th}}$ layer to trivially make use of input-side sparsity, since all-zero features can be safely skipped even for zero-padded layers. Because all convolutions can exploit both input- and output-side sparsity, the speed-up gained from pruning is quadratic with respect to the pruning ratio. For instance, dynamically pruning half of the channels in all layers gives rise to a dynamic CNN that uses approximately $\frac{1}{4}$ of the original MACs.

Theoretically, FBS does not introduce the reparameterisation discussed above. By batch normalizing the convolution output, the convolution kernel $\boldsymbol{\theta}_l$ is invariant to scaling. Computationally, it is more efficient to train. Many alternative methods use non-differentiable π functions that produce on/off decisions. In general, DNNs with these policy functions are incompatible with SGD, and resort to reinforcement learning for training. In contrast, FBS allows end-to-end training, as wta is a piecewise differentiable and continuous function like ReLU. Srivastava *et al.* [153] suggested that in general, a network is easier and faster to train for complex tasks and less prone to catastrophic forgetting, if it uses functions such as wta that promote local competition between many sub-networks.

3.3.3 Evaluating FBS

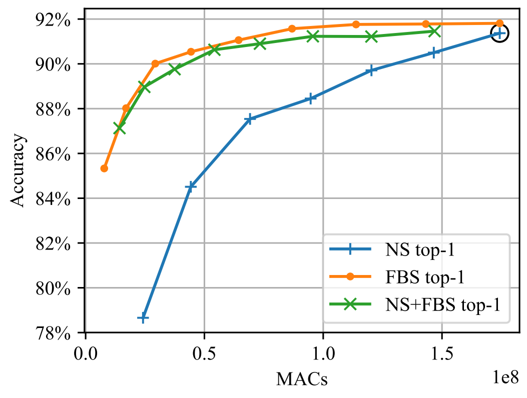
I use CIFAR10 [94] and ImageNet [93], the two popular image classification datasets, to evaluate FBS. I designed a custom CNN for CIFAR10 named CifarNet based on the VGG class of networks (see Table 3.3 for its structure), using only 1.3 million parameters with 91.37% and 99.67% top-1 and top-5 accuracies respectively. CifarNet is much smaller than a VGG16 on CIFAR10 [114], which uses 20 million parameters and is only 2.29% more accurate, it is basically a variant of VGG9 with Batch Normalization and Dropout. Because of its compactness, CifarNet is more challenging to accelerate. By faithfully reimplementing *Network Slimming* (NS) [114], a popular static coarse-grained pruning method, I closely compare FBS with NS under various speedup constraints. For ImageNet, I augment two popular CNN variants, ResNet18 and [66] and VGG16 [151], and provide a detailed accuracy/MACs trade-off comparison against recent structured pruning and dynamic execution methods.

I trained CifarNet (see Table 3.3 for its architecture) with a 0.01 learning rate and a 256 batch size. We reduced the learning rate by a factor of 10× for every 100 epochs. To compare FBS against NS fairly, every model with a new target MACs budget were consecutively initialized with the previous model, and trained for a maximum of 300 epochs, which is enough for all models to converge to the best obtainable accuracies. For NS, we follow [114] and start training with an ℓ^1-norm sparsity regularization weighted by 10^{-5} on the BN scaling factors. We then prune at 150 epochs and fine-tune the resulting network without the sparsity regularization.

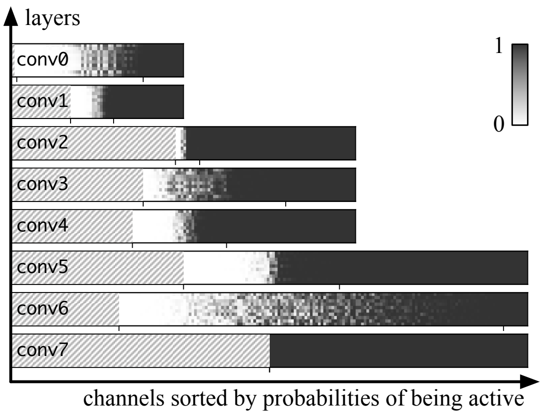
We additionally employed image augmentation procedures from [93] to preprocess each training example. Each CIFAR10 example was randomly horizontal flipped and slightly perturbed in the brightness, saturation and hue.

ImageNet classifiers, were trained with a procedure similar to the one above. The difference was that they were trained for a maximum of 35 epochs, the learning rate was decayed for every 20 epochs, and NS models were all pruned at 15 epochs. For image preprocessing, we additionally cropped and stretched/squeezed images randomly following [93].

The proposed FBS method begins by first replacing all convolutional layer computations with Equation (3.6), and initializing the new convolutional kernels with previous parameters. Initially, I do not suppress any channel computations by using density $d = 1$ in Equation (3.7) and fine-tune the resulting network. For fair comparison against NS, I then follow [114] by iteratively decrementing the overall density d of the network by 10% in each step, and thus gradually using fewer channels to sweep the accuracy/performance trade-off. The difference is that NS prunes channels by ranking globally, while FBS prunes around $1 - d$ of each layer.



(a) CifarNet accuracy/MACs trade-off



(b) Channel skipping probabilities

Figure 3.4: Experimental results on CifarNet. I compare in (a) the accuracy/MACs trade-off between FBS, NS and FBS+NS. The baseline is emphasized by the circle ◯. The heat map in (b) reveals the individual probability of skipping a channel for each channel (x-axis), when an image of a category (y-axis) is shown to the network with $d = 1$.

By respectively applying NS and FBS to CifarNet classifier and incrementally increasing sparsity, I produce the trade-off relationships between number of operations (measured in MACs) and the classification accuracy as shown in Figure 3.4a. FBS clearly surpasses NS in its ability to retain the task accuracy under an increasingly stringent computational budget. Besides comparing FBS against NS, I am interested in combining both methods, which demonstrates the effectiveness of FBS if the model is already less redundant, *i.e.* it cannot be pruned further using NS without degrading the accuracy by more than 1%. The composite method (NS+FBS) is shown to successfully regain most of the lost accuracy due to NS, producing a trade-off curve closely matching FBS. It is notable that under the same 90.50% accuracy constraints, FBS, NS+FBS, and NS respectively achieve 3.93×, 3.22×, and 1.19× speed-up ratios. Conversely for a 2× speed-up target, they respectively produce models with accuracies not lower than 91.55%, 90.90% and 87.54%.

Figure 3.4b uses 8 heat maps to respectively represent the channel skipping probabilities of the 8 convolutional layers. The brightness of the pixel at location (x, y) denotes the probability of skipping the x^{th} channel when looking at an image of the y^{th} classification category. The heat maps verify my belief that the auxiliary network learned to predict which channels specialise to which features, as channels may have drastically distinct probabilities of being used for images of different categories. The model here is a CifarNet using FBS with $d = 0.5$, which has a top-1 accuracy of 90.59% (top-5 99.65%). Moreover, channels in the heat maps are sorted so the channels that are on average least frequently evaluated are placed on the left, and channels shaded in stripes are never evaluated. The network in Figure 3.4b is not only approximately 4× faster than the original, by removing the unused channels, it also reduces the number of weights by 2.37×. This reveals that FBS naturally subsumes channel pruning strategies such as NS, as I can simply prune away channels that are skipped regardless of the input. It is notable that even though I specified a universal density d, FBS learned to adjust its dynamicity across all layers, and prune different ratios of channels from the convolutional layers.

I also resent a detailed layer-wise study of CifarNet in Table 3.3 on the CIFAR10 classification task. The CifarNet, a custom designed CNN, with less than 1.30 million parameters and takes 174 million MACs to perform inference for a 32-by-32 RGB image. The architecture is illustrated in Table 3.3, where all convolutional layers use 3×3 kernels, the **Shape** column shows the shapes of each layer's features, and `pool7` is a global average pooling layer. Table 3.3 additionally provides further comparisons of layer-wise compute costs between FBS, NS, and the composition of the two methods (NS+FBS). It is notable that the FBS column has two different output channel counts, where the former is the number of computed channels for each inference, and the latter is the number of channels remaining in the layer after removing the unused channels.

Residual networks [66] adopt sequential structure of residual blocks: $\mathbf{x}_b = K(\mathbf{x}_{b-1}) + F(\mathbf{x}_{b-1})$, where $\mathbf{x}_b$ is the output of the b^{th} block, K is either an identity function or a downsampling convolution, and F consists of a sequence of convolutions. For residual networks, I directly apply FBS to all convolutional layers, with a difference in the way I handle the feature summation. Because the $(b+1)^{\text{th}}$ block receives as input the sum of the two features with sparse channels $K(\mathbf{x}_{b-1})$ and $F(\mathbf{x}_{b-1})$, the channels of this sum is considered sparse only when the same channels in both features are sparse.

By applying FBS and NS respectively to ResNet18, I saw the ImageNet validation accuracy of FBS consistently outperforms NS under different speed-up constraints. For speed-up, I refer to the number of multiply accumulates required to finish an inference run

Layer	**Shape**	**Number of MACs (Output Channels)**			
		Original	NS	FBS	NS+FBS
`conv0`	30×30	1.5M (64)	1.3M (52)	893K (32/62)	860K (32)
`conv1`	30×30	33.2M (64)	27.0M (64)	8.4M (32/42)	10.2M (39)
`conv2`	15×15	16.6M (128)	15.9M (123)	4.2M (64/67)	5.9M (74)
`conv3`	15×15	33.2M (128)	31.9M (128)	8.3M (64/79)	11.6M (77)
`conv4`	15×15	33.2M (128)	33.1M (128)	8.3M (64/83)	12.1M (77)
`conv5`	8×8	14.1M (192)	13.4M (182)	3.6M (96/128)	4.9M (110)
`conv6`	8×8	21.2M (192)	11.6M (111)	5.4M (96/152)	4.3M (67)
`conv7`	8×8	21.2M (192)	12.3M (192)	5.4M (96/96)	4.5M (116)
`pool7`	1×1				
`fc`	1×1	1.9K (10)	1.9K (10)	960 (10)	1.1K (10)
Total		174.3M	146.5M	44.3M	54.2M
Saving		-	1.19×	3.93×	3.21×

Table 3.3: The network structure of CifarNet for CIFAR10 classification. In addition, we provide a detailed per-layer MACs compariso between FBS, NS, and the composition of them (NS+FBS). I minimise the models generated by the three methods while maintaining a classification accuracy of at least 90.5%.

compared to the original dense model. For instance, at $d = 0.7$, it utilises only 1.12 billion multiply accumulates (MACs) (1.62× speed-up) to achieve a top-1 error rate of 31.54%, while NS requires 1.51 billion MACs (1.21× faster) for a similar error rate of 31.70%. When compared across recent dynamic execution methods examined in Table 3.4, FBS demonstrates simultaneously the highest speed-up and the lowest error rates. It is notable that the baseline accuracies for FBS refer to a network that has been augmented with the auxiliary layers featuring FBS but suppress no channels, *i.e.* $d = 1$. FBS brings immediate accuracy improvements, an increase of 1.73% in top-1 and 0.46% in top-5 accuracies, to the baseline network, which is in line with my observation on CifarNet.

Method	**Dynamic**	**Baseline**		**Accelerated**		**Speed-up**
		Top-1	Top-5	Top-1	Top-5	
Soft Filter Pruning [69]		29.72	10.37	32.90	12.22	1.72×
Network Slimming ([114], my implementation)		31.02	11.32	32.79	12.61	1.39×
Discrimination-aware Channel Pruning [213]		30.36	11.02	32.65	12.40	1.89×
Low-cost Collaborative Layers [39]	✓	30.02	10.76	33.67	13.06	1.53×
Channel Gating Neural Networks [75]	✓	30.98	11.16	32.60	12.19	1.61×
Feature Boosting and Suppression (FBS)	✓	**29.29**	**10.32**	**31.83**	**11.78**	**1.98**×

Table 3.4: Comparisons of error rates of the baseline and accelerated ResNet18 models.

Since VGG16 is computationally intensive with over 15 billion MACs, I first applied NS on this large network to reduce the computational and memory requirements, and this eases the training of the FBS-augmented variant. I assigned a 1% budget in top-5 accuracy degradation and compressed the network using NS, which gave us a smaller VGG16 with

Method	Dynamic	Δ top-5 errors (%) 3×	4×	5×
Filter Pruning ([105], reproduced by [70])		—	8.6	14.6
Perforated CNNs [46]		3.7	5.5	—
Network Slimming ([114], my implementation)		1.37	3.26	5.18
Channel Pruning [70]		0.0	1.0	1.7
AMC [71]		—	—	1.4
ThinNet-Conv [116]		0.37	—	—
Feature Boosting and Suppression (FBS)	✓	0.04	**0.52**	**0.59**

Table 3.5: Comparisons of top-5 error rate increases for VGG16 on ImageNet validation set under 3×, 4× and 5× speed-up constraints. The baseline has a 10.1% top-5 error rate. Results from He *et al.* [70] only show numbers with one digit after the decimal point.

20% of all channels pruned. The resulting network is a lot less redundant, which almost halves the compute requirements, with only 7.90 billion MACs remaining. Table 3.5 shows the performance of different structured pruning and dynamic execution methods to FBS. At a speed-up of 3.01×, FBS shows a minimal increase of 0.44% and 0.04% in top-1 and top-5 error rates respectively. At a 5.23× speed-up, it only degrades the top-1 error rate by 1.08% and the top-5 by 0.59%.

3.4 Focused quantisation for custom hardware

3.4.1 Quantising sparse and dense models

Pruning algorithms compress CNNs by setting weights to zero, thus removing connections or neurons from the models. In particular, fine-grained pruning [112, 57] provides the best compression by removing connections at the finest granularity, *i.e.* individual weights. Quantisation methods reduce the number of bits required to represent each value, and thus further provide memory, bandwidth and compute savings. *Shift quantisation* of weights, which quantises weight values in a model to powers-of-two or zero, *i.e.* $\{0, \pm1, \pm2, \pm4, \ldots\}$, is of particular of interest, as multiplications in convolutions become much-simpler bit-shift operations or simply wiring in certain cases. The computational cost in hardware can thus be significantly reduced without a detrimental impact on the model's task accuracy [207]. Fine-grained pruning is often in conflict with quantisation, as pruning introduces various degrees of sparsities to different layers [125]. Linear quantisation methods (integers) have uniform quantisation levels and non-linear quantisations (logarithmic, floating-point and shift) have fine levels around zero but levels grow further apart as values get larger in magnitude. Both linear and nonlinear quantisations thus provide precision where it is not actually required in the case of a pruned CNN. It is observed that empirically, very few non-zero weights concentrate around zero in some layers that are sparsified with fine-grained pruning (see Figure 3.5c for an example). Shift quantisation is highly desirable as it can be implemented efficiently, but it becomes a poor choice for certain layers in sparse models, as most near-zero quantisation levels are under-utilized (Figure 3.5d).

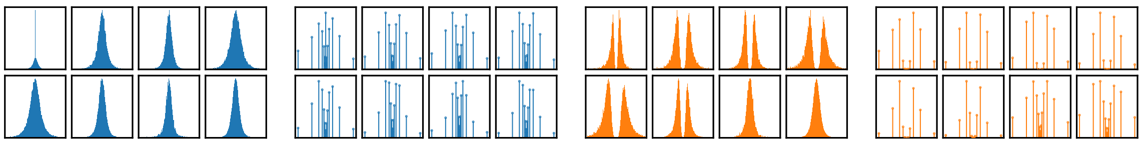

(a) Dense layers (b) After shift quan (c) Sparse layers (d) After shift quan

Figure 3.5: The weight distributions of the first 8 layers of ResNet18 on ImageNet. (a) shows the weight distributions of the layers, (c) similarly shows the distributions (excluding zeros) for a sparsified variant. (b) and (d) respectively quantise the weight distributions on the left with 5-bit shift quantisation. Note that in some sparse layers, greedy pruning encourages weights to avoid near zero values. Shift quantisation on these layers thus results in poor utilisation of the quantisation levels.

This dichotomy prompts the question, *how can we quantise sparse weights efficiently and effectively?* Here, efficiency represents not only the reduced model size but also the minimised compute cost. Effectiveness means that the quantisation levels are well-utilised. From an information theory perspective, it is desirable to design a quantisation function Q such that the quantised values in $\hat{\boldsymbol{\theta}} = Q(\boldsymbol{\theta})$ closely match the prior weight distribution.

3.4.2 Focused quantisation

Shift quantisation constrains weight values to be either zero or powers-of-two. A representable value in a $(\mathsf{k}+2)$-bit shift quantisation is given by:

$$v = s \cdot 2^{e-\mathsf{b}}, \tag{3.8}$$

where $s = \{-1, 0, 1\}$ denotes either zero or the sign of the value, e is an integer bounded by $[0, 2^{\mathsf{k}} - 1]$, and b is the bias, a layer-wise constant which scales the magnitudes of quantised values. I use $\hat{\boldsymbol{\theta}} = Q^{\mathsf{shift}}_{n,b}[\boldsymbol{\theta}]$ to denote a n-bit shift quantisation with a bias b of a weight value $\boldsymbol{\theta}$ to the nearest representable value $\hat{\boldsymbol{\theta}}$.

Intuitively, it is desirable to concentrate quantisation effort on the high probability regions in the weight distribution in sparse layers. Focused quantisation $Q_{\mathrm{focus}}[\boldsymbol{\theta}]$ is designed specifically for this purpose, and applied in a layer-wise fashion.

$$Q[\boldsymbol{\theta}] = z_\theta \alpha \sum_{c \in C} \delta_{c,\mathrm{m}_{\boldsymbol{\theta}}} \mathrm{Q}^{\mathrm{rec}}[\boldsymbol{\theta}], \text{ where } \mathrm{Q}^{\mathrm{rec}}_c[\boldsymbol{\theta}] = Q^{\mathsf{shift}}_{n,b}\left[\frac{\boldsymbol{\theta} - \mu_c}{\sigma_c}\right] \sigma_c + \mu_c. \tag{3.9}$$

Here $z_{\boldsymbol{\theta}}$ is a predetermined constant $\{0, 1\}$ binary value to indicate if $\boldsymbol{\theta}$ is pruned, and it is used to set pruned weights to 0. The set of components $c \in C$ determines the locations to focus quantisation effort, each specified by the component's mean μ_c and standard deviation σ_c. The Kronecker delta $\delta_{c,\mathrm{m}_{\boldsymbol{\theta}}}$ evaluates to either 1 when $c = \mathrm{m}_{\boldsymbol{\theta}}$, or 0 otherwise. In other words, the constant $\mathrm{m}_{\boldsymbol{\theta}} \in C$ chooses which component in C is used to quantise $\boldsymbol{\theta}$. Finally, $\mathrm{Q}_c^{\mathrm{rec}}[\boldsymbol{\theta}]$ locally quantises the component c with shift quantisation. Following Zhu *et al.* [212] and Leng *et al.*[102], I additionally introduce a layer-wise learnable scaling factor α initialised to 1, which empirically improves the task accuracy.

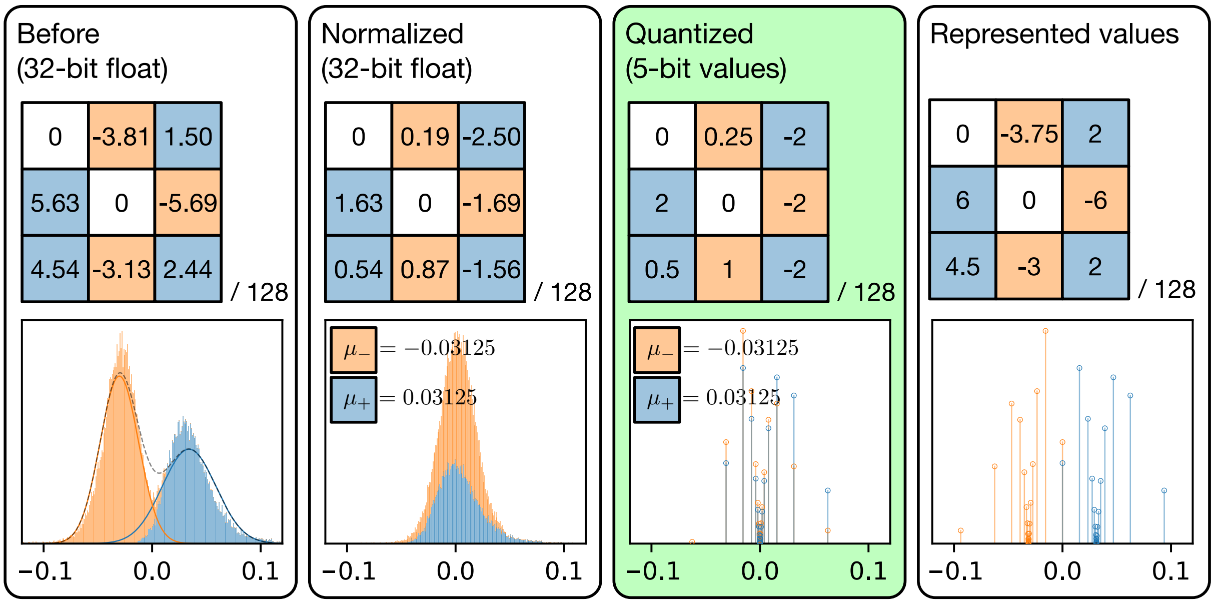


Figure 3.6: The step-by-step process of recentralised quantisation of unpruned weights on `block3f/conv1` in sparse ResNet-50. Each step shows how it changes a filter and the distribution of weights. Higher peaks in the histograms denote values found with higher frequency. Values in the filter share a common denominator 128, indicated by "/128". The first estimates the high-probability regions with a Gaussian mixture, and assign weights to a Gaussian component. The second normalises each weight. The third quantises the normalised values with shift quantisation and produces a representation of quantised weights used for inference. The final block visualises the actual numerical values after quantisation.

Hyperparameters μ_c and σ_c in recentralised quantisation can be optimised by applying the following two-step process in a layer-wise manner, which first identifies regions with high probabilities (first block in Figure 3.6), then locally quantise them with shift quantisation (second and third blocks in Figure 3.6). First, in general, the weight distribution resembles a mixture of Gaussian distributions. It is thus more efficient to find a Gaussian mixture model $Q^{\mathrm{mix}}(\boldsymbol{\theta})$ to approximate the original distribution $p(\boldsymbol{\theta}|\mathcal{D})$, where $\mathcal{D}$ represents the dataset, to closely optimise the above objective:

$$Q^{\mathrm{mix}}(\boldsymbol{\theta}) = \sum_{c \in C} \lambda_c f(\boldsymbol{\theta}|\mu_c, \sigma_c), \tag{3.10}$$

where $f(\boldsymbol{\theta}|\mu_c, \sigma_c)$ is the probability density function of the Gaussian distribution $N(\mu_c, \sigma_c)$, the non-negative λ_c defines the mixing weight of the c^{th} component and $\Sigma_{c \in C} \lambda_c = 1$. Here, I find the set of hyperparameters μ_c, σ_c and λ_c contained in $\boldsymbol{\phi}$ that maximises Q^{mix} given that $\boldsymbol{\theta} \sim p(\boldsymbol{\theta}|\mathcal{D})$. This is known as the *maximum likelihood estimate* (MLE), and the MLE can be efficiently computed by the *expectation-maximisation* (EM) algorithm [36]. In practice, it is sufficient to use two Gaussian components, $C = \{-, +\}$, for identifying high-probability regions in the weight distribution. For faster EM convergence, I initialize μ_-, σ_- and μ_+, σ_+ respectively with the means and standard deviations of negative and positive values in the layer weights respectively, and λ_-, λ_+ with $\frac{1}{2}$. I then generate $m_{\boldsymbol{\theta}}$ from the mixture model, which individually selects the component to use for each weight. For this, $m_{\boldsymbol{\theta}}$ is evaluated for each $\boldsymbol{\theta}$ by sampling a categorical distribution where the probability of assigning a component c to $m_{\boldsymbol{\theta}}$, *i.e.* $p(m_{\boldsymbol{\theta}} = c)$, is $\lambda_c f(\boldsymbol{\theta}|\mu_c, \sigma_c)/Q^{\text{mix}}(\boldsymbol{\theta})$.

Finally, I set the constant b to a powers-of-two value, chosen to ensure that $Q^{\text{shift}} n, b\,[\cdot]$ allows at most a proportion of $\frac{1}{2^n+1}$ values to overflow and clips them to the maximum representable magnitude. In practice, this heuristic choice makes better use of the quantisation levels provided by shift quantisation than disallowing overflows. After determining all of the relevant hyperparameters with the method described above, $\hat{\boldsymbol{\theta}} = Q[\boldsymbol{\theta}]$ can be evaluated to quantise the layer weights.

As I have discussed earlier, the weight distribution of sparse layers may not always have multiple high-probability regions. It is possible that the sparse layers have two high probability masses that are overlapping. Under this scenario, I can simply use n-bit shift quantisation. To decide whether to use shift or recentralised quantisation, it is necessary to introduce a metric to compare the similarity between the pair of components. While the KL-divergence provides a measure for similarity, it is however non-symmetric, making it unsuitable for this purpose. To address this, I propose to first normalise the distribution of the mixture, then to use the 2-Wasserstein metric between the two Gaussian components after normalisation as a decision criterion, which I call the *Wasserstein separation*:

$$\mathcal{W}(c_1, c_2) = \frac{1}{\sigma^2}\left((\mu_{c_1} - \mu_{c_2})^2 + (\sigma_{c_1} - \sigma_{c_2})^2\right), \tag{3.11}$$

where μ_c and σ_c are respectively the mean and standard deviation of the component $c \in \{c_1, c_2\}$, and σ^2 denotes the variance of the entire weight distribution. FQ can then adaptively pick to use recentralised quantisation for all sparse layers except when $\mathcal{W}(c_1, c_2) < w_{\text{sep}}$, and shift quantisation is used instead. I found $w_{\text{sep}} = 2.0$ usually provides a good decision criterion. In the next subsection, I additionally study the impact of quantising a model with different w_{sep} values.

3.4.3 Evaluating focused quantisation

Table 3.6 compares the accuracy and compression rates before and after applying the focused quantisation compression (FC) pipeline under different quantisation bit-widths. It demonstrates the effectiveness of FC on the models. Sparsified ResNets with 7-bit weights are at least 16× smaller than the original dense model with marginal degradation (≤0.24%) in top-5 accuracy. MobileNets, which are much less redundant and more compute-efficient models to begin with, achieved a smaller CR at around 8× and slightly larger accuracy degradations (≤0.89%). Yet when compared to the ResNet-18 models, it is not only more accurate, but also has a significantly smaller memory footprint at 1.71 MB.

In Table 3.7, I compare FC with many state-of-the-art model compression schemes. It shows that FC simultaneously achieves the best accuracy and the highest CR on both ResNets. Trained Ternary Quantisation (TTQ) [212] quantises weights to ternary values, while INQ [207] and extremely low bit neural network (denoted as ADMM) [102] quantise weights to ternary or powers-of-two values using shift quantisation. Distillation and Quantisation (D&Q) [131] quantise parameters to integers via distillation. Note that D&Q's results used a larger model as baseline, hence the compressed model has high accuracy and low CR. I also compared against Coreset-Based Compression [42] comprising pruning, filter approximation, quantisation and Huffman encoding. This method exploits inter-filter dependencies in the CNN filter banks using corest representations and prune unimportant filters. For ResNets, I additionally compare against ThiNet [115], a filter pruning method, and Clip-Q [158], which interleaves training steps with pruning, weight sharing and quantisation. FC again achieves the highest CR (18.08×) and accuracy (74.86%).

Table 3.6: The accuracy (%), sparsities (%) and CRs of focused compression on ImageNet models. The baseline models are dense models before compression and use 32-bit floating-point weights, and 5 bits and 7 bits denote the number of bits used by individual weights of the quantised models before Huffman encoding.

Model	Top-1	Δ	Top-5	Δ	Sparsity	Size (MB)	CR (×)
ResNet-18	68.94	-	88.67	-	0.00	46.76	-
Pruned	69.24	0.30	89.05	0.38	74.86	8.31	5.69
5 bits	68.36	-0.58	88.45	-0.22	74.86	2.86	16.33
7 bits	68.57	-0.37	88.53	-0.14	74.86	2.94	15.92
ResNet-50	75.58	-	92.83	-	0.00	93.82	-
Pruned	75.10	-0.48	92.58	-0.25	82.70	11.76	7.98
5 bits	74.86	-0.72	92.59	-0.24	82.70	5.19	18.08
7 bits	74.99	-0.59	92.59	-0.24	82.70	5.22	17.98
MobileNet-V1	70.77	-	89.48	-	0.00	16.84	-
Pruned	70.03	-0.74	89.13	-0.35	33.80	6.89	2.44
7 bits	69.13	-1.64	88.61	-0.87	33.80	2.13	7.90
MobileNet-V2	71.65	-	90.44	-	0.00	13.88	-
Pruned	71.24	-0.41	90.31	-0.13	31.74	5.64	2.46
7 bits	70.05	-1.60	89.55	-0.89	31.74	1.71	8.14

Table 3.7: Comparisons of top-1 and top-5 accuracies (%) and CRs with various compression methods. Numbers with * indicate results not originally reported and calculated by us. Note that D&Q used a much larger ResNet18, the 5 bases used by ABC-Net denote 5 separate binary convolutions. LQ-Net used a "pre-activation" ResNet-18 [67] with a 1.4% higher accuracy baseline than ours.

ResNet-18	Top-1	Top-5	Size (MB)	CR ($\times$)
TTQ [212]	66.00	87.10	2.92*	16.00*
INQ (2 bits) [207]	66.60	87.20	2.92*	16.00*
INQ (3 bits) [207]	68.08	88.36	4.38*	10.67*
ADMM (2 bits) [102]	67.0	87.5	2.92*	16.00*
ADMM (3 bits) [102]	68.0	88.3	4.38*	10.67*
ABC-Net (5 bases, or 5 bits) [110]	67.30	87.90	7.30*	6.4 *
LQ-Net (preact, 2 bits) [192]	68.00	88.00	2.92*	16.00*
D&Q (large) [131]	**73.10**	**91.17**	21.98	2.13*
Coreset [42]	68.00	-	3.11*	15.00
Focused compression (5 bits, sparse)	68.36	88.45	**2.86**	**16.33**
ResNet-50	Top-1	Top-5	Size (MB)	CR ($\times$)
INQ (5 bits) [207]	74.81	92.45	14.64*	6.40*
ADMM (3 bits) [102]	74.0	91.6	8.78*	10.67*
ThiNet [115]	72.04	90.67	16.94	5.53*
Clip-Q [158]	73.70	-	6.70	14.00*
Coreset [42]	74.00	-	5.93*	15.80
Focused compression (5 bits, sparse)	**74.86**	**92.59**	**5.19**	**18.08**

As mentioned previously, I use the Wasserstein distance $\mathcal{W}(c_1, c_2)$ between the two components in the Gaussian mixture model as a metric to differentiate whether recentralised or shift quantisation should be used. It is important to understand the choices of this introduced hyperparameter. In my experiments, I specified a threshold $w_{\text{sep}} = 2.0$ such that for each layer, if $\mathcal{W}(c_1, c_2) \geq w_{\text{sep}}$ then recentralised quantisation is used, otherwise shift quantisation is employed instead. Figure 3.7 shows the impact of choosing different w_{sep} ranging from 1.0 to 3.5 at 0.1 increments on the Top-1 accuracy. This model is a CIFAR10 [94] classifier with only 9 convolutional layers, so that it is possible to repeat training 100 times for each w_{sep} value to produce high-confidence results. Note that the average validation accuracy is minimised when the layer with only one high-probability region uses shift quantisation and the remaining 8 use recentralised quantisation, which verifies my intuition.

3.5 Applying compression techniques to hardware

It is obvious that pruning reduces model sizes and the number of FLOPs (floating point operations). Another important impact of pruning is on the memory side, especially

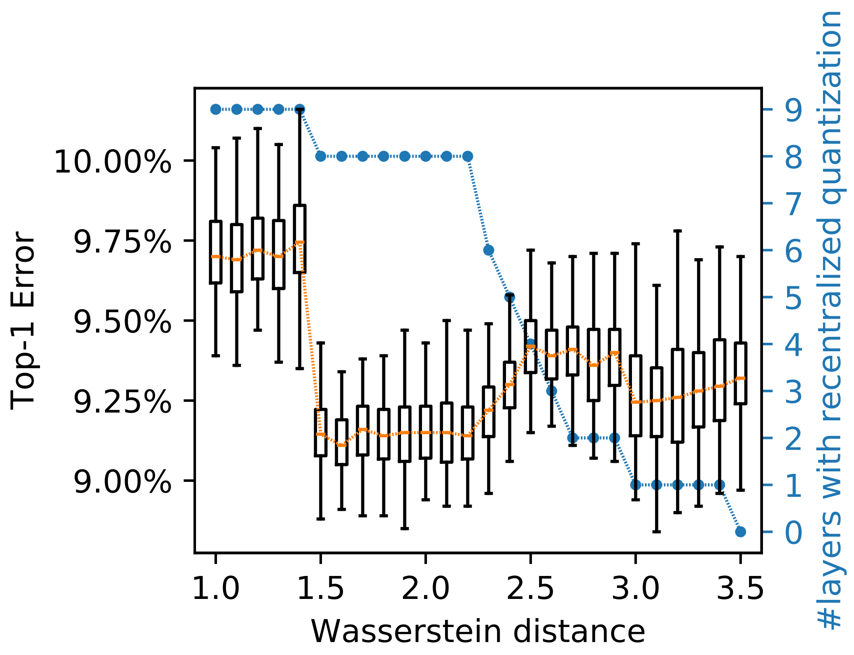


Figure 3.7: The effect of different threshold values on the Wasserstein distance. The larger the threshold, the fewer the number of layers using recentralised quantisation instead of shift quantisation.

if the inputs are batched. Not only does FBS use much fewer FLOPs, it also further demonstrates significant reductions in bandwidth and memory requirements of hardware platforms. In low-powered edge computing, inference is typically performed on a single image. In Table 3.8, I observe a large reduction in the number of memory accesses required to load weights and read/write activations, as a convolution with FBS uses sparse input and evaluates sparse output activations. Since these read/write operations are often carried out by DRAM accesses, minimising them directly translates to power-savings as I spend much less energy performing expensive DRAM operations. Table 3.8 further reveals that in diverse application scenarios such as low-end edge devices and cloud-based GPU platforms, the peak memory usages by the optimised models are also significantly smaller than the originals, which in general improves cache utilisation.

Table 3.9: Comparison of the original ResNet-18 with successive quantisations applied on weights, activations and BN parameters. Each row denotes added quantisation on new components.

Quantised	Top-1	Δ	Top-5	Δ
Baseline	68.94	-	88.67	-
+ Weights (5-bit FQ)	68.36	-0.58	88.45	-0.22
+ Activations (8-bit integer)	67.89	-1.05	88.08	-0.59
+ BN (16-bit integer)	67.95	-0.99	88.06	-0.61

Quantising weights using FC can significantly reduce the arithemtic computation

Table 3.8: Comparisons of the number of memory accesses and peak memory utilisation of VGG16 and ResNet18 on ImageNet with FBS respectively under 3× and 2× inference speed-up constraints. The **Weights** and **Activations** columns respectively show the total amount of weight and activation accesses required by all convolutions for a single image inference. The **Peak Memory Usage** columns show the peak memory usages by the same models with different batch sizes.

Model	**Total Memory Accesses**		**Peak Memory Usage**	
	Weights	Activations	Edge (1 image)	Cloud (128 images)
VGG16	56.2 MB	86.5 MB	24.6 MB	3.09 GB
VGG16 3×	23.9 MB (2.35×)	40.8 MB (2.12×)	9.97 MB (2.47×)	1.24 GB (2.47×)
ResNet18	44.6 MB	17.8 MB	9.19 MB	0.47 GB
ResNet18 2×	20.5 MB (2.18×)	12.3 MB (1.45×)	4.68 MB (1.96×)	0.31 GB (1.49×)

Table 3.10: Computation resource estimates of custom accelerators for inference assuming the same compute throughput.

Configuration	#Gates	Ratio
ABC-Net (5 bases, or 5 bits)	806.1 M	2.93×
LQ-Net (2 bits)	314.4 M	1.14×
Shift quantisation (3 bits, unsigned)	275.2 M	1.00×
FQ (5 bits)	275.6 M	1.00×
FQ (5 bits) + Huffman	276.4 M	1.00×

complexity in models. By further quantising activations and BN parameters to integers, the expensive floating-point multiplications and additions in convolutions can be replaced with simple bit-shift operations and integer additions. This can be realised with much faster hardware implementations using custom hardware, which directly translates to energy saving and much lower latencies. In Table 3.9, I evaluate the impact on accuracies by progressively applying FQ on weights, and integer quantisations on activations and batch normalisation (BN) parameters.

Figure 3.8 shows an accelerator design of the dot-products used in the convolutional layers with recentralised quantisation for inference. Using this, in Table 3.10, I provide an estimated logic usage required by the implementation to compute a convolution layer with 3×3 filters with a padding size of 1, which takes as input a $8 \times 8 \times 100$ activation and produces a $8 \times 8 \times 100$ tensor output. Additionally, I compare FQ to shift quantisation, ABC-Net [110] and LQ-Net [192]. The #Gates indicates the lower bound on the number of two-input logic gates required to implement the custom hardware accelerators for the convolution, assuming an unrolled architecture and the same throughput. Internally, a 5-bit FQ-based inference uses 3-bit unsigned shift quantised weights, with a minimal overhead for the added logic. Scaling constants σ_- and σ_+ are equal and thus can be fused into α_l. Perhaps most surprisingly, a 5-bit FQ has more quantisation levels yet uses fewer logic gates, when compared to ABC-Net and LQ-Net implementing the same

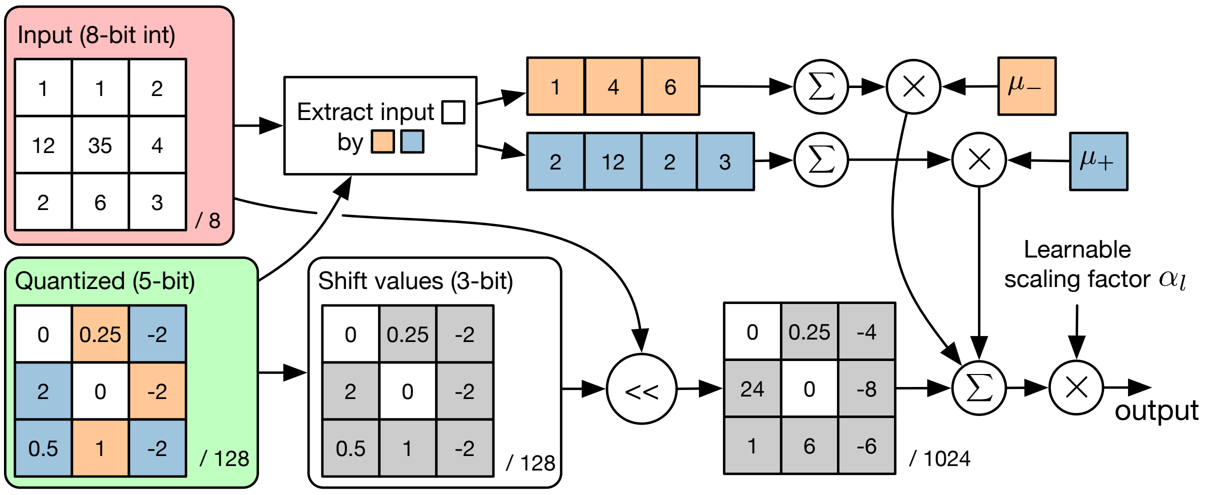


Figure 3.8: An implementation of the dot-product used in convolution between an integer input and a filter quantised by recentralised quantisation. The notation $/N$ means the filter values share a common denominator N.

convolution but with different quantisations. Both ABC-Net and LQ-Net quantise each weight to N binary values, and compute N parallel binary convolutions for each binary weight. The N outputs are then accumulated for each pixel in the output feature map. Even with the optimal compute pattern proposed by the two methods, there are at least $O(MN)$ additional high-precision multiply-adds, where M is the number of parallel binary convolutions, and N is the number of output pixels. This overhead is much more significant when compared to other parts of compute in the convolution. As shown in Table 3.10, both have higher logic usage because of the substantial amount of high-precision multiply-adds. In contrast, FQ has only one final learnable layer-wise scaling multiplication that can be further optimised out as it can be fused into BN for inference. Despite having more quantisation levels and a higher CR, and being more efficient in hardware resources, the fully quantised ResNet-18 in Table 3.9 can still match the accuracy of a LQ-Net ResNet-18.

3.6 Summary

In this chapter, I showed how combining basic compression algorithms can provide multiplicative compression gains (Section 3.2). Although the same phenomenon has been observed in Deep Compression, I revealed that there is further compression optimisation opportunities if allowing a more flexible combination of pruning and quantisation strategies. In Section 3.3 and Section 3.4, I introduced novel compression techniques for pruning and quantisation respectively. These two novel compression techniques exploit certain characteristics of the neural networks in order to achieve a better compression quality. The first technique helps the models to go beyond the state-of-the-art compression ratios by observing that channel saliencies are different for various inputs. The second method exploits the post-pruning weights distributions to build a quantisation scheme. Finally, I further explained how these novel compression methods can be used on real hardware and what do they mean for general-purpose and custom hardware platforms.

Chapter 4

Multi-precision multi-arithmetic CNN hardware accelerators for efficient neural networks

In this chapter, I will look at how re-designing the hardware can improve the performance of DNN inference. Hardware designs often limit the number of possible software compression techniques (*e.g.* pruning and quantisations) that can be applied to the hardware. This chapter then explores how a more flexible hardware deign can allow us to apply a group of more sophisticated compression techniques.

I will first provide an overview of different styles of DNN accelerators and then propose an alternative design that focuses on pipelining a number of small compute cores. I show how this hardware design can be integrated with software optimisations, offering a high throughput, low latency design on FPGAs without the need for off-chip memory traffic. Later, I will discuss the computation flow and data reuse strategies of the proposed hardware architecture in details and also demonstrate an automated co-design flow for generating accelerators on custom computing devices.

This chapter includes relevant contents published in:

- Yiren Zhao, Xitong Gao, Xuan Guo, Junyi Liu, Erwei Wang, Robert Mullins, Peter YK Cheung, George Constantinides, and Cheng-Zhong Xu. Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs. In *2019 International Conference on Field-Programmable Technology, pages 45–53. IEEE, 2019 (ICFPT 2019)*

Yiren Zhao conceived and designed the experiments. Yiren Zhao implemented a test version of a multi-arithmetic DNN accelerator in high level synthesis (HLS), Junyi Liu provided correction and testing in this process. Erwei Wang changed and deployed the test version to a more detailed implementation in high level synthesis. Yiren Zhao later realised that the HLS tool was limiting the performance of deeply pipeline designs and produced a hardware description in a RTL (SystemVerilog) design. Xuan Guo later further optimised the implementation. Yiren Zhao and Xitong Gao worked on extending the software framework Mayo to help the hardware generation.

4.1 Accelerator designs for neural networks: homogeneous systolic arrays vs streaming cores

Hardware that uses a homogeneous large systolic array currently dominates the design of CNN accelerators; a great number of parallel multiplication-adds are used as a large compute core and both weights and activations are buffered in on-chip memory [199, 25]. The large systolic array is time-shared as a number of different convolutions or fully-connected layers reuse the same hardware. This approach is popular in ASIC accelerators and can exploit data parallelism to achieve high throughput. In terms of hardware, the potential design space of such a homogeneous core is large, since there is a wide variety of data reuse patterns available as illustrated in Chapter 2. It is then hard to guarantee that a single set of hardware optimisation is optimal to a wide range of neural network types. From the software point of view, the efficiency of a systolic array accelerator relies on data reuse and this complicates the compilation stack. The compiler has to carefully schedule workloads to maximise the utilisation of the computation units while ensuring a good data reuse to reduce the number of off-chip memory accesses [23]. This process is non-trivial due to the ever-changing input data sizes, channel counts, kernel sizes and ever-emerging neural network operators [197].

The systolic array structure is a popular DNN accelerator paradigm on both ASIC and FPGA devices. A fully-custom specialised ASIC is a very attractive solution in terms of energy efficiency. However, a good efficiency and performance may be limited to the applications used to tune the architecture and may quickly degrade if the accelerator has to run networks with different characteristics (standard limits to specialisation) [85]. On the other hand, FPGA designs have great fine-grained reconfigurability, but normally run at a lower clock frequency compared to ASIC accelerators. An alternative design principle for DNN acceleration is to rely heavily on hardware reconfigurability and the ability to customise heavily pipelined cores. This design approach is more suitable for FPGA devices. In this way, the design complication in hardware increases significantly but each core has the flexibility to be optimised solely for its targeting neural network layer. The computation of a network can also be divided and pipelined into a number of smaller compute cores (so-called flattened streaming cores). Each computation core, streamed by activations, is only responsible for the calculations of individual layers [48, 198] to maximise efficiency.

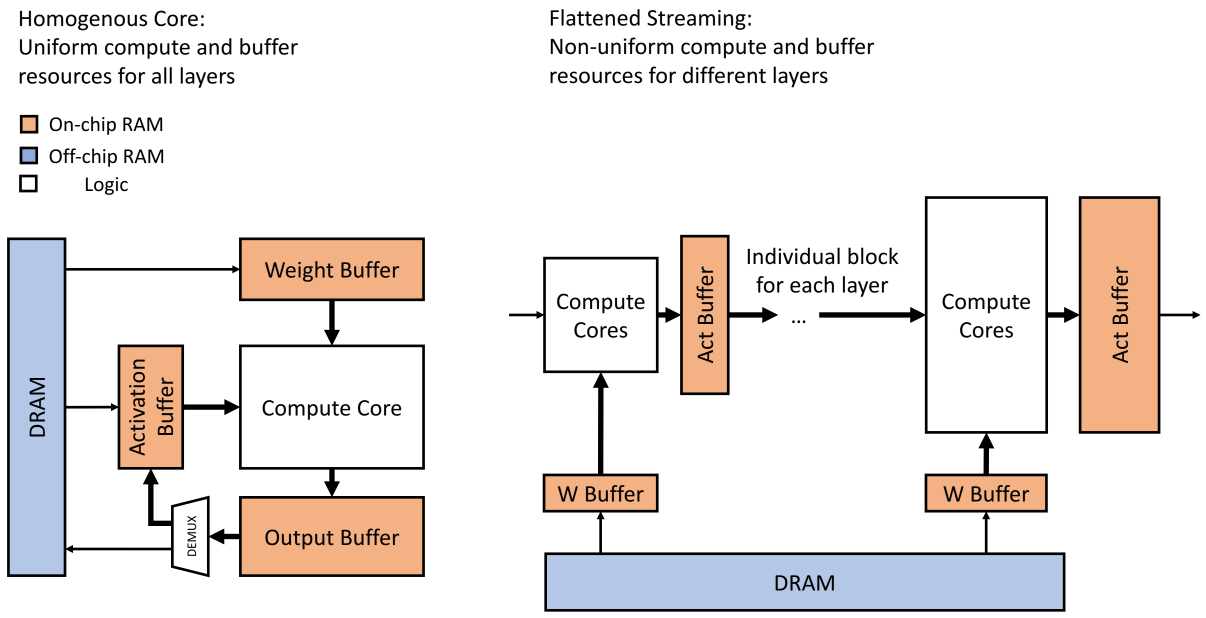


Figure 4.1: An illustration of a homogeneous core (left) and flattened streaming cores (right).

Figure 4.1 shows a graphical illustration of a homogeneous core and flattened streaming cores. The homogeneous core has a large compute unit that utilises most of the design budget, whereas the flattened streaming cores design uses a number of smaller cores. It is also worth to mention that, in the flatten streaming cores design, since the throughput is determined by the slowest stage, there exists opportunities for fast (normally later) computation stages to be unrolled less to save resources without having any impacts on the system throughput. I will discuss this unrolling strategy in details in Section 4.5.3.

4.2 Low precision inference in hardware

The GPU has to make a decision about what number formats or arithmetic to support when designing the hardware, while the FPGAs can be adapted and programmed to have various hardware implementations. GPUs today can support narrow data types, but don't have as much freedom over the bit-widths and arithmetics, and other opportunities such as mixing arithmetic. Yet DNNs are, in general, often over-provisioned and inherently redundant; this makes low-precision quantisation an essential technique to drastically reduce the memory consumed by the network's parameters, and even allows DNN inference to be computed entirely with low-cost arithmetic operations, rather than floating-point ones. Many works [32, 81] train CNNs to use low-precision weights and activations with minimal accuracy loss, while others pushed the limit by using ternarised weights $\{0, \pm 1\}$ [82, 104, 212], and even constraining both weights and activations to binary values $\{\pm 1\}$ [80]. However, binarised and ternarised CNNs struggle to achieve state-of-the-art accuracies on large datasets. The most popular quantisation strategies used for DNN accelerators are fixed-point and shift quantisations, since they achieve a good trade-off between model accuracy and hardware efficiency, I have discussed these two number representation systems in Chapter 2.

FPGA-based accelerators generally uniformly apply one of the above low precision quantisation methods [148, 55]. This specialisation provides efficiency and performance gains when compared to GPUs, but does not fully exploit the reconfigurability of such devices. Alternatively, we could consider selecting the most appropriate quantisation and quantisation hardware on a per layer basis. The aim here to is both minimise the impact of quanisation loss and maximise the model task accuracy. My discussion in Section 3.2 showed that different layers are suitable for different quantisation strategies due to the differences in weights distributions. In addition, using different quantisations on for different computing cores dedicated for each layer will reduce the hardware circuit area. The idea of allowing different bit-width for different layers is not new. Bit-serial accelerators [142, 143] target exactly this problem as they provide scope to optimise away superfluous computation at the bit-level when computing with fixed-point numbers. These accelerators will run at a much higher clock frequency compared to their bit-parallel competitors to compensate for the increased number of clock cycles to finish multiplications. However, the use of bit-serial operations is still a limited exploration of adaptive arithmetic in the fixed-point number territory, later in this chapter, I will illustrate a workflow that automatically generate accelerators that not only have computing cores using different bit-widths but also different arithmetics.

4.3 The Tomato framework

The framework is designed to target *Convolutional Neural Networks* and to support different styles of convolutional and fully-connected layers. The tool can be extended with support for additional layer types and may also serve as a skeleton for automatic hardware accelerator generations in other machine learning domains. The objective of this framework is to design an easy-to-use hardware-software co-design flow with the flexibility of supporting different arithmetics and numerical precisions on a per-layer basis.

Figure 4.2 shows Tomato and Mayo at a high level. The Tomato framework is at the hardware level and can be easily integrated with the existing Mayo framework. Mayo can produce compressed models and these models with then be used in Tomato to generate a hardware accelerator Figure 4.3 further shows the details of hardware accelerator generation flow named Tomato. The framework starts with an automated design process in software which uses a search algorithm to explore the choices of fixed-point and shift arithmetics with varying precisions on the pre-trained CNN model. It then produces optimised models

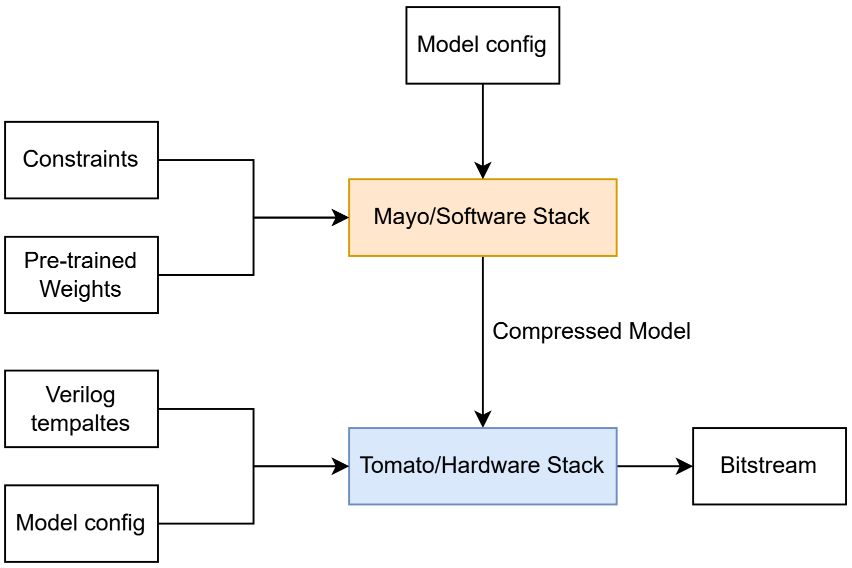


Figure 4.2: A high level illustration of how Mayo and Tomato can be integrated to produce directly a bitstream for FPGA devices.

that are fully quantised while satisfying the accuracy constraints. In the exploration procedure, it iteratively uses an accurate hardware resource estimator to provide rapid predictions of hardware costs and minimise the costs for the searched models. I designed an analytical model for predicting the hardware utilisation based on my empirical data, the designed prediction model then shows a highly accurate estimation (within 2% errors) on the targeting FPGA devices. The cost, latency, number of lookup tables (LUTs) and block RAMs (BRAM) usage, is estimated using analytical models generated from post synthesis results for a wide range of module parameters. The final optimised CNN model is then fine-tuned on the original training dataset to minimise accuracy degradation.

It is notable that from the optimised model, Tomato generates dedicated compute engines for each convolutional layer in a streaming fashion. As I have mentioned earlier, the compute engines are connected in a pipeline, each takes a stream of inputs and produces a stream of outputs. The isolated compute engines can thus have the freedom to use different quantisations with individual bit-widths. To minimise hardware utilisation, layers that exceed throughput requirements can be folded (i.e. only partially unrolled) to share individual processing elements temporally. The framework automatically computes the optimal unroll factors required to parallelise each layer which minimises stalls and idle circuits. Finally, the framework generates SystemVerilog output describing the hardware implementation of the input model, which is in turn synthesised into circuits with fine-tuned weights.

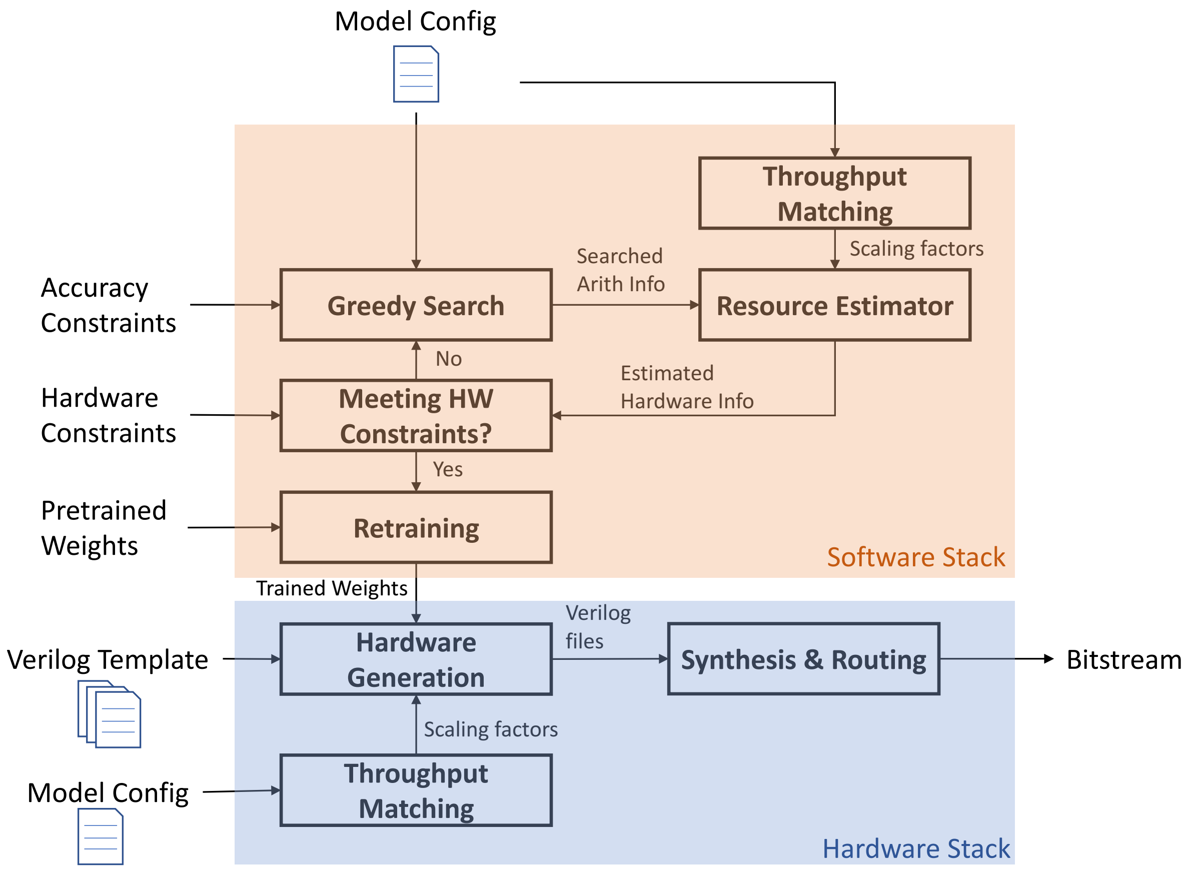


Figure 4.3: Framework overview for generating the accelerator for a targeting network on a particular dataset. The overall framework is composed of the Software Stack (Mayo) and Hardware Stack (Tomato).

4.4 Automatic quantisation exploration

As both the bit-width of weights and their representation (*i.e.* either shift or fixed-width) may vary on a layer-by-layer basis, it is intractable to explore all possible combinations exhaustively. For this reason, I introduce an algorithm which minimises the hardware design complexity under a given accuracy constraint. Algorithm 1 provides an overview of the search algorithm, which accepts as inputs a CNN model with weight parameters θ and N layers $\{l_1, l_2, \ldots, l_N\}$, the accuracy constraint α_{budget}, the hardware resource constraint h_{budget}, and an initial state of quantisation hyperparameters q_0 which uses 8-bit fixed-point quantisation for all layers. Here, $\theta', \alpha \leftarrow \text{finetune}(\theta, q, E)$ fine-tunes the model parameters θ under hyperparameters q for E epochs and returns the validation accuracy of fine-tuned model. I found empirically $E = 3$ is sufficient to recover most accuracy loss due to quantisation. To traverse the search space efficiently, I introduce a relation $R(L)$, where L is a set of modifiable layers. Each transition $(q, q') \in R(L)$ finds a one step change to the configuration q, *i.e.* decreasing the bit-width by 1 or changing the arithmetic used by a layer $\text{layer\_changed}(q, q') \in L$. In each step, the algorithm is designed to greedily find a new configuration q' from q which results in the steepest reduction in hardware resources $\text{hwcost}(q) - \text{hwcost}(q')$ until all layers cannot be modified further without violating the accuracy constraint. Additionally, if the hardware resource constraint $\text{hwcost}(q) \leq h_{\text{budget}}$ is already satisfied then I exit early to minimise accuracy loss. In my experiments, I chose α_{budget} to be 0.95α, where α is the original accuracy, to generate a fully quantised model with efficient hardware usage. The resulting model is then fine-tuned to further increase accuracy.

Algorithm 1 Search Algorithm

```
 1: function SEARCH($\theta, q_0, \alpha_{\text{budget}}, h_{\text{budget}}, E$)
 2:     $q \leftarrow q_0$; $L \leftarrow \{l_1, l_2, \dots, l_N\}$
 3:     while $L \neq \varnothing$ do
 4:         $q' \leftarrow \text{argmax}(q, q') \in R(L)(\text{hwcost}(q) - \text{hwcost}(q'))$
 5:         $\theta', \alpha \leftarrow \text{finetune}(\theta, q', E)$
 6:         if $\alpha \geq \alpha_{\text{budget}}$ then
 7:             $q \leftarrow q'$
 8:             $\theta \leftarrow \theta'$
 9:             if $\text{hwcost}(q') \leq h_{\text{budget}}$ then
10:                 break
11:             end if
12:         else
13:             $L \leftarrow L - \text{layer\_changed}(q, q')$
14:         end if
15:     end while
16:     return $q, \theta$
17: end function
```

This quantisation search is the core for the software part in the automatic generation flow as illustrated as the orange regions in Figure 4.2.

4.5 Automatic hardware generation

4.5.1 Macro-architectures

From the macro-architecture point of view, Figure 4.4 shows the differences between a normal homogeneous core style accelerator and the generated flattened streaming cores. In the flattened streaming cores, each convolution has its dedicated compute engine, slide buffer and weight buffer. Since the hardware is generated solely for the targeted CNN and each compute core is dedicated for a particular layer, with a suitable strategy to parallelise compute, the generated hardware can reach very high compute efficiency and have minimal idle hardware. In fact, in later measurements, I will show that the compute units generated from Tomato has a high utilisation that is constantly at around 84%.

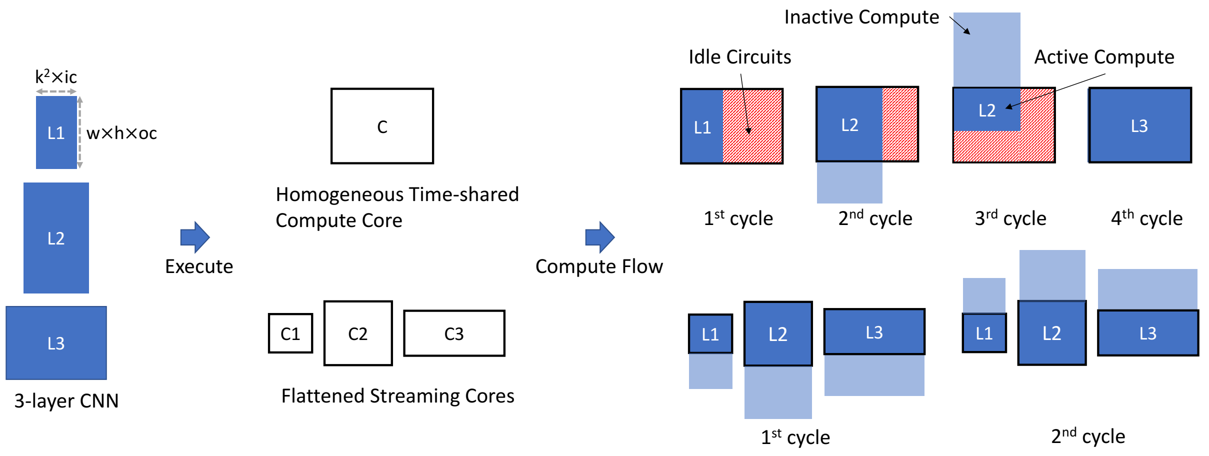


Figure 4.4: An illustration of the computation flow on executing three layers of convolutions (L_1, L_2, L_3) at different clock cycles. C represents a large homogeneous compute core and C_1, C_2, C_3 are smaller compute cores in a flattened streaming architecture. The rectangle block of each convolution layer represents input dimensions of a convolution flattened in 2D. k, ic, oc, w, h are kernel size, input/output channels, width and height of input feature feature maps respectively.

4.5.2 Micro-Architecture: roll-unrolled convolutions

In this section, I introduce the roll-unrolled convolution compute core, this is designed to minimise hardware costs when input and output data rates permit. As an example, I consider a convolution layer with a kernel size of K, which takes as input feature maps $\mathbf{X}_{l-1}$ of shape $H \times W \times C_{l-1}$, and produces output feature maps $\mathbf{X}_l$ with H' and W' depending on the stride size and padding length. In addition, it is noteworthy that a convolution with a stride size of 1 can produce pixels in the output feature maps at the same rate of it taking input pixels. A convolution with a stride size of 2, however, produces an output pixel 4 times slower than it can consume an input pixel. Layers in a convolutional network can therefore process their feature maps at an exponentially slower rate as more proceeding layers are strided, and in turn have greater opportunities to reuse data-paths. By way of illustration, assuming the input image is fed at a rate of 1 cycle per pixel, the input/output throughput rates of each layer in a MobileNet-V1 can be found in the last column of Table 4.1.

In order to maximise a layer's utilisation and minimise hardware costs, rather than introducing stall cycles, I introduce two unroll factors, U and U', for input and output channels respectively. I partially roll input channel dimension C_{l-1} into U-sized blocks to save hardware resource. I still accumulate C_l values in parallel. In other words, all C_l channels of a pixel of the output feature maps are unrolled and computed concurrently. Fully unrolling output channels during multiplication and accumulation is essential to allow stall-free computation. Finally, output channels are rolled after accumulation to U'-sized blocks for batch normalisation. Fused batch normalisation and ReLU operations are time-shared for U' output channels, as the next layer has an input block size equal to U'. As it processes all input channels C_{l-1} in blocks of size U, it uses only $U \times C_l$, instead of $C_{l-1} \times C_l$ parallel shift-accumulate or multiply-accumulate units, requiring $\left\lceil \frac{C_{l-1}}{U} \right\rceil$ cycles to complete the computation of a single pixel of all output channels, as shown in Figure 4.5.

Tomato does not use roll-unrolled in depthwise convolutions. Figure 4.6 shows the computation pattern for depthwise convolutions. In contrast to normal convolutions, depthwise convolutions are channel-wise operations, *i.e.* they do not exchange information across channels. By rolling input channels in depthwise layers, the generated outputs are also rolled, different from the normal roll-unrolled compute pattern. In this way, I exactly match the throughput of incoming and upcoming computations while minimising resource utilisation. Each parallel adder tree sums up K^2 values and is fully pipelined, where K is the kernel size.

Roll-unrolled should not be confused with loop tiling. Loop tiling reorders the access pattern so that it is more friendly to either cache access or DRAM bandwidth utilisation in systolic array based CNN accelerators. As Tomato pipelines multiple frames instead of batching them, I did not change the access pattern. The purpose of rolling and unrolling in Tomato's streaming cores are to minimise hardware and provide a stall-free computation dataflow, a fundamentally different objective.

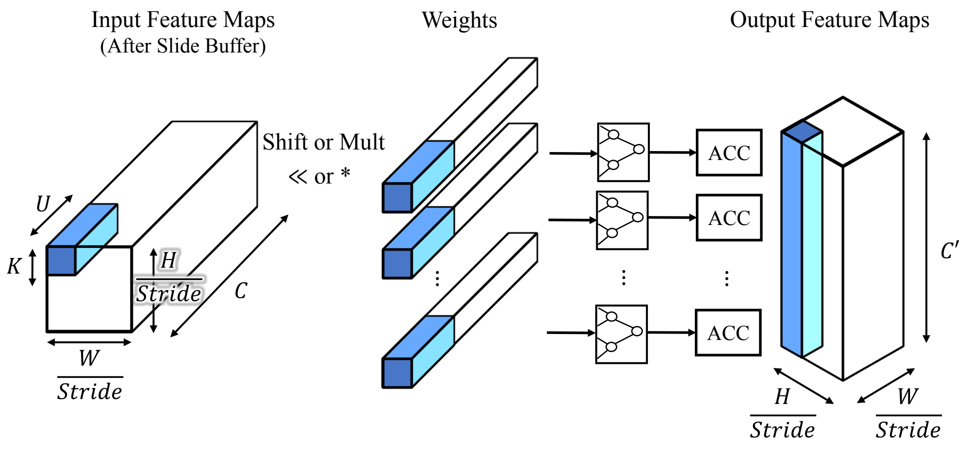


Figure 4.5: An illustration of roll-unrolled computation for normal convolution (including pointwise convolution): blue indicates data elements computed in a single cycle.

4.5.3 Striding and rolling: matching the throughput

By adjusting the unroll factors U and U', the framework smartly matches the throughput between convolution layers with different channel counts and stridings for higher efficiency. The only free parameter now is the input pixel rate at the very first layer of the CNN. The input pixel rate determines how many pixels of an input image are fed into the accelerator at each clock cycle. For instance, an input rate of $\frac{1}{32}$ means I consume 1 input

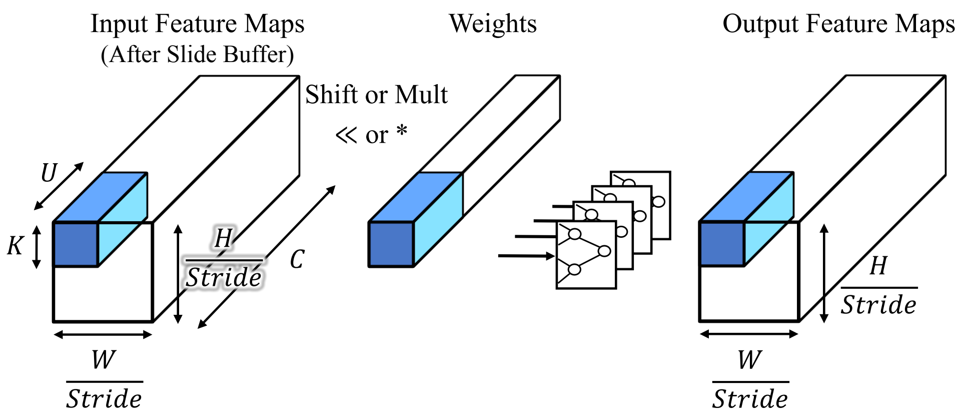


Figure 4.6: An illustration of depthwise convolution: blue indicates data elements computed in a single cycle.

pixel in 32 clock cycles. The choice of the input pixel rate directly impacts the trade-off between performance and the hardware resources required. If this input pixel rate is 1, the generated hardware is optimised for performance, fully-pipelined, and never stalls the input pixel stream. When the input pixel rate decreases, because of the automatic matching, unroll factors of all subsequent convolution layers decrease and the generated hardware thus utilises fewer resources but has an increased latency.

I now explain how the automated throughput matching works. The framework utilises the classic sliding window design — one pixel of a output feature map is produced once all pixels of the sliding window on input feature maps have arrived [12]. The input stream and output stream of strided convolutions, however, can have different input and output rates. For instance, when the stride size is 2, the output stream is then 4 times slower than the input stream (striding occurs in the two spatial dimensions). Table 4.1 shows the unrolling factors U and U' that the framework picked for each convolution in MobileNet-V1 when choosing the input pixel rate to be 1. Here, for each pixel, $\frac{C_i}{U}$ represents the number of clock cycles required to iterate over all input channel values, and $\frac{C_o}{U'}$ is the number of cycles required to finish generating all output channel values. Taking the second depthwise convolution layer as an example, this layer has a stride size of 2 and the framework rolls computations on the output channel side by a factor of $\frac{C_o}{U'} = 16$ so that $U' = 4$ values of each output channel are computed concurrently. Finally, all of the unrolling information is provided to the hardware templates in order to instantiate the appropriate hardware.

4.5.4 Batch normalisation and rounding

Each convolved output has an inflated precision as mentioned in the previous section, and I then subsequently apply batch normlisation on these values.

Batch normalisation (BN) is commonly used in CNNs to accelerate training [83]. As shown in Equation (4.1), during inference, BN normalises convolutional outputs $\mathbf{X}_l$ in

Table 4.1: Unrolling factors U and U' are generated by the throughput matcher for MobileNet-V1, depending on the input and output channel counts (C_{l-1}, C_l), and the stride of each convolution. dw and pw are depthwise and pointwise convolution. s1 and s2 represents strides are 1 and 2.

Types	$\mathbf{C_{l-1}}$ / $\mathbf{C_l}$	$\mathbf{U}$ / $\mathbf{U'}$	$\frac{\mathbf{C_{l-1}}}{\mathbf{U}}$ / $\frac{\mathbf{C_l}}{\mathbf{U'}}$
Conv / s2	3 / 32	3 / 8	1 / 4
Conv dw / s1	32 / 32	8 / 8	4 / 4
Conv pw / s1	32 / 64	8 / 16	4 / 4
Conv dw / s2	64 / 64	16 / 4	4 / 16
Conv pw / s1	64 / 128	4 / 8	16 / 16
Conv dw / s1	128 / 128	8 / 8	16 / 16
Conv pw / s1	128 / 128	8 / 8	16 / 16
Conv dw / s2	128 / 128	8 / 2	16 / 64
Conv pw / s1	128 / 256	2 / 4	64 / 64
Conv dw / s1	256 / 256	4 / 4	64 / 64
Conv pw / s1	256 / 256	4 / 4	64 / 64
Conv dw / s2	256 / 256	4 / 1	64 / 256
Conv pw / s1	256 / 512	1 / 2	256 / 256
Conv dw / s1	512 / 512	2 / 2	256 / 256
Conv pw / s1	512 / 512	2 / 2	256 / 256
Conv dw / s2	512 / 512	2 / 1	256 / 512
Conv pw / s1	512 / 1024	1 / 1	512 / 1024
Conv dw / s1	1024 / 1024	1 / 1	1024 / 1024
Conv pw / s1	1024 / 1024	1 / 1	1024 / 1024
Avg Pool / s1	1024 / 1024	1 / 1	1024 / 1024
FC / s1	1024 / 1000	1 / 1	1024 / 1000

a channel-wise fashion with a moving mean μ and a moving standard deviation σ, then applies affine transformation on them with the learned γ and β:

$$\mathbf{X_l} = \gamma \frac{\mathbf{X}_l - \mu}{\sigma} + \beta \tag{4.1}$$

It is notable that Equation (4.1) can be re-arranged into a channel-wise affine transformation. In the CNN feed-forward stages, I respectively quantise the scaling and offset factors of this affine transformation to fixed-point numbers:

$$\mathbf{y} = \text{quantise}_{8.8}\left(\frac{\gamma}{\sigma}\right)\mathbf{x} + \text{quantise}_{8.8}\left(\beta - \frac{\gamma\mu}{\sigma}\right), \tag{4.2}$$

where $\text{quantise}_{8.8}(\mathbf{z})$ quantises $\mathbf{z}$ into 16-bit fixed-point values with a binary point at 8.

4.6 Evaluating the performance of automatically generated hardware

4.6.1 Experimental setups

In this section, I report results for automatically generated hardware implementations for three distinct networks, each optimised for a different dataset. I use CifarNet [201], an 8-layer CNN with 1.30M parameters and 174M multiply-accumulates on the CIFAR-10 dataset [94], MobileNet-V1 [73] on the ImageNet dataset [93] and a customised 5-layer CNN (FashionNet) for the Fashion MNIST dataset [176]. The first two networks are relatively large, but the last network is small. I use MobileNet-V1 design as a comparison to showcase the performance achieved from this hardware and software co-design workflow in comparison to other published accelerators. The hardware part (SystemVerilog output) is generated automatically using templates by the framework. Synplify Premier DP is used for synthesis and post-synthesis timing analysis. The final designs are actually implementable on FPGAs using Xilinx Vivado to place and route the full-size MobileNet design.

4.6.2 Resource utilisation

For MobileNet, the design is fully-pipelined and never stalls the input stream ($\frac{\#\text{OPs}}{\#\text{OPs/cycle}} = 224 \times 224$). Note that 224×224 is the input image size and this means the accelerator consumes an entire image in exactly 224×224 clock cycles. The design utilises 84% of my 13479 compute units (shift-and-add or multiply-and-add) on every clock cycle. The high utilisation rate of the hardware translates to a high activity ratio in the circuits since most components are active all the time. This fully quantised MobileNet found by Algorithm 1 uses 3-bit shift weights in its pointwise layers, and fixed-point weights in its depthwise layers with precisions ranging from 3 to 7.

Table 4.2: Summary of tested networks on the Tomato. IPP stands for input pixel rate.

Network	IPP	Platform	Perf Metrics			LUTs	Registers	BRAMs	DSPs		Top-1	Top-5	Size
MobileNet-V1	1	Intel Stratix 10	**Frequency**	156 MHz	**Used**	926K	583K	1430	297	**Orig.**	70.71	89.53	33.92MB
			Latency	358us	**Total**	1866K	3732K	11721	5760	**Quant.**	68.02	88.02	16.1MB
			Throughput	3109 fps	**Ratio**	49%	15%	12%	5%	**Δ**	-2.69	-1.51	2.11×
CifarNet	$\frac{1}{32}$	Intel Stratix V	**Frequency**	207MHz	**Used**	304K	280K	771	84	**Orig.**	91.37	99.68	4.94MB
			Latency	261us	**Total**	469K	939K	2.56K	256	**Quant.**	91.06	99.58	520KB
			Throughput	6317 fps	**Ratio**	64%	29%	30%	32%	**Δ**	-0.31	-0.10	9.73×
CifarNet	$\frac{1}{288}$	Intel Cyclone V	**Frequency**	116MHz	**Used**	102K	84.7K	715	82	**Orig.**	91.37	99.68	4.94MB
			Latency	4.01ms	**Total**	227K	454K	1.22K	342	**Quant.**	91.06	99.58	520KB
			Throughput	393 fps	**Ratio**	44%	18%	58%	24%	**Δ**	-0.31	-0.10	9.73×
FashionNet	$\frac{1}{9}$	Xilinx Artix 7	**Frequency**	98MHz	**Used**	49.3K	32.7K	40	240	**Orig.**	91.79	99.67	443KB
			Latency	138us	**Total**	63.4K	127K	135	240	**Quant.**	91.57	99.56	65.3KB
			Throughput	13.9kfps	**Ratio**	77%	25%	29%	100%	**Δ**	-0.22	-0.11	6.78×

Table 4.2 shows the total amount of hardware utilised for the generated accelerators for all networks on different devices. The MobileNet design uses an input pixel rate of 1 for

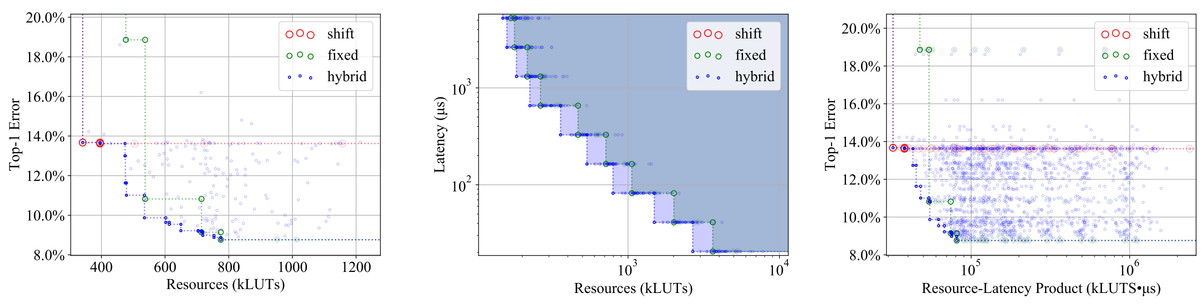
the best performance (achieving 3109 FPS on an Intel Stratix 10). The proposed workflow is a scalable one since it can adjust the input pixel rate to control a trade-off between performance and hardware utilisation. CifarNet results in Table 4.2 show how it is possible to target a small FPGA device (Cyclone V) by adjusting this factor to $\frac{1}{288}$. The results suggest a 3× reduction in LUT usage compared to the design when the input pixel rate is set to $\frac{1}{32}$. It is also observed an increase in latency, but part of the increase attributes to the frequency differences running on various devices. On the other hand, if provided with a small network (FashionNet), the proposed framework generates hardware that classifies at a latency of 0.14ms on a very small FPGA device. The quantised FashionNet uses 3-bit shift quantisation in the most resource-intensive third layer, and the remaining layers use fixed-point weights with bit-widths from 5 to 7. Although FashionNet is small, it is a good example of a specialised network produced for resource constrained edge devices; other examples include emotion recognition [19].

I explore in Figure 4.7 the optimised CifarNets obtained with Algorithm 1 (denoted by the "hybrid" points) and compare the results against shift ("shift") and fixed-point ("fixed") models with all layers sharing the same bit-widths ranging from 3 to 8. To explore the trade-off between top-1 error rates and resources, I ran Algorithm 1 20 times by respectively taking as inputs the accuracy budget values α_{budget} ranging from 80% to 100% at 1% increments. Here h_{budget} is set to 0 as I always minimise the resource utilisation. I constrain each layer to use either shift or fixed-point quantised weights and choose a bit-width ranging from 3 to 8. Additionally, $E = 0$ meaning that I skip the fine-tuning process; without fine-tuning the accuracies are sub-optimal but the search process above can complete within 1 hour. Figure 4.7a shows the trade-off between resource utilisation and top-1 errors under the same throughput constraints. Figure 4.7b further varies the throughput scaling of the optimised results, and shows that when synthesised into circuits, the the optimised models ("hybrid") consistently outperforms models ("shift" and "hybrid") with either shift or fixed-point quantisation under the same bit-width applied across all layer weights. Finally, Figure 4.7c illustrates all results found by the three methods and the trade-off relationship between top-1 error rates and resource-latency products.

Hybrid quantisation reduces the accuracy degradation and improves model compression rates by utilising multi-precision multi-arithmetic convolutions. Importantly, I consider the ImageNet [93] classification task for MobileNet. This challenging dataset leaves less headroom for compression techniques. The classification results achieved on this large dataset using a relatively compact network proves that the workflow is also robust on other cases where the model is over-provisioned on the target dataset.

Table 4.3: A comparison of CNN inference performance on FPGA and GPU platforms. The quantisation of weights and activations are on the left. Target platform, frequency, latency, throughput and arithmetic performance are on the right. Metrics with * are our best-case estimations as they are not mentioned in the original papers. Note VGG16 has a similar top-5 accuracy to MobileNet-V1 when neither is quantised, many of the works below do not report ImageNet accuracies after quantisation.

		Quantisation(s)						
	Implementation	Weights	Acts	Platform	Frequency (MHz)	Latency (ms)	Throughput (FPS)	Arithmetic perf. (GOP/s)
VGG16	Throughput-Opt [155]	FXP8	FXP16	Intel Stratix V	120	262.9	3.8*	117.8
	fpgaConvNet [164]	FXP16	FXP16	Xilinx Zynq XC7Z045	125	197*	5.07	156
	Angel-Eye [55]	BFP8	BFP8	Xilinx Zynq XC7Z045	150	163*	6.12*	188
	Going Deeper [132]	FXP16	FXP16	Xilinx Zynq XC7Z045	150	224*	4.45	137
	Shen *et al.* [147]	FXP16	FXP16	Xilinx Virtex US XCVU440	200	49.1	26.7	821
	HARPv2 [122]	BIN	BIN	Intel HARPv2	–	8.77*	114	3500
	GPU [122]	FP32	FP32	Nvidia Titan X	–	–	121	3590
MobileNet	Ours	Mixed	FXP8	Intel Stratix 10	156	0.32	3109	3536
	Ours	Mixed	FXP8	Xilinx Virtex US+ XCVU9P	125	0.40	2491	2833
	Zhao *et al.* [199]	FXP16	FXP16	Intel Stratix V	200	0.88	1131	1287
	Zhao *et al.* [200]	FXP8	FXP8	Intel Stratix V	150	4.33	231	264
	GPU	FP32	FP32	Nvidia GTX 1080Ti	–	279.4	515	586



(a) Number of LUTs vs top-1 error under the same $\frac{1}{32}$ scaling and the same throughput.

(b) Number of LUTs vs latency for models with $\leq 10\%$ top-1 errors.

(c) Error vs area-latency product for all optimized models.

Figure 4.7: A case study of trade-off options among hardware utilization (LUTs), performance (latency) and model accuracy (top-1 error rate) before fine-tuning, targeting a clock frequency of 200 MHz. The LUTs and latency numbers are from the hardware estimator. Here, "shift" and "fixed" respectively indicate using the same shift and fixed-point quantisation method across all layers with the same weight precisions. The "hybrid" points are optimized by Algorithm 1. The area shaded in red, green and blue respectively denote the 2D Pareto frontier of "shift", "fixed" and "hybrid" optimized results.

4.6.3 Performance evaluation

I compare the MobileNet-V1 design generated by my framework with existing FPGA accelerators in Table 4.3. This comparison only considers networks in the ImageNet dataset that achieve greater than 70% top-1 accuracy when not quantised. The computer vision community spends a significant amount of effort in optimizing model architecture, I note that it is important to generate results for the latest models as they offer the best accuracy/cost trade-offs. Results for older models in terms of GOp/s can be misleading.

The generated design is different from most existing designs examined in Table 4.3 in the following ways. First, the framework exploits hybrid quantisation to minimise the impact of quantisation errors. Second, using the throughput scaling trick, the amount of hardware required is reduced significantly. Most of the examined designs rely on a high DRAM bandwidth with a large monolithic compute core. As discussed previously, a large compute core cannot explore multi-precision and multi-arithmetic quantisations and struggles to fully utilise compute units on convolutions with varying channel counts, kernel sizes and input feature sizes. Tomato generated designs compute various layers concurrently and quantise each layer differently, thus achieving a very high utilisation of my compute units and to operate at around 3.5 TOps/s on Stratix 10. Note that, for accelerators that I compare against, the arithmetic performance reported in Table 4.3 considers the peak performance assuming an unbounded DRAM bandwidth. In reality, such designs can easily be limited by the available memory bandwidth. In contrast, this is not a concern in this design as all weights and activations are held on the FPGA. Additionally, the generated design has a high throughput since operations rarely stall. Designs proposed by Bai *et al.* [12] and Zhao *et al.* [199] have to execute computations in a layer-wise fashion, and thus operations in the next layer only executes when the current layer finishes. In this framework, similar to Shen *et al.* [148], computations in different layers happen concurrently in the same pipeline stage while later layers never stall earlier layers. Moreover, consecutive image inputs can be fully pipelined, because the generated hardware utilises streaming sliding windows. These features help the hardware to achieve high throughput compared to other designs (Table 4.3). The proposed workflow avoids complex and time-consuming design space exploration as necessary in many compared FPGA accelerators [12, 200].

In terms of performance, this design achieves a higher throughput and a lower latency compared to all designs listed in Table 4.3. I notice that most CNN accelerators report *theoretical upper bounds* for arithmetic performance and throughput. In terms of latency, the numbers are reported optimistically assuming DRAM accesses cause no stalls. In this design, since I stream in pixels of the input image, the computation pattern differs from most existing designs. The reported values in Table 4.3 represents the true performance and make no assumptions regarding DRAM bandwidth. The system automatically produces an implementation of MobileNet for the Stratix 10 FPGA that outperforms Zhao *et al.* [199] by 2.44× in latency and 3.52× in throughput.

4.6.4 Multi-FPGA acceleration

The flattened streaming style employed by Tomato makes it easy to partition the generated design across multiple FPGAs This feature makes Tomato highly scalable with respect to network sizes and/or FPGA sizes. I demonstrate in this section an example of partitioning

MobileNet-V1 onto two Stratix V FPGAs, connected through enhanced small form-factor pluggable (SFP+) interfaces. I present the performance results in comparison to Zhang *et al.* [190] in Table 4.4, and the detailed hardware utilisation information in Table 4.5. The latency is not penalised thanks to the low latency of SFP+, which contributes only a 0.0013ms latency overhead.

The simple case study of partitioning the same MobileNet-V1 design to two devices demonstrates that, first, Tomato generated designs are scalable from single to multi-FPGAs; second, aiming accelerating new network architectures with mixed quantisations bring significant improvements in accuracies, latency and throughput.

Table 4.4: Multi-FPGA acceleration of CNNs. MBNet represents MobileNet, VGG-D and VGG-E are both VGG16 based networks but different configurations, one is latency oriented and one is throughput oriented [190].

Network	**Acc (%)**	**#Device**	**Lat (ms)**	**Tpt (GOPs)**
MbNet-V1 (ours)	68.02	2 Stratix V	0.32	3536
VGG-D [190]	66.52	2 VX690t	200.9	203
VGG-E [190]	66.51	7 VX690t	151.8	1280

Table 4.5: Multi-FPGA hardware utilization.

Device No	**Frequency**	**LUTs**	**Regs**	**BRAM**	**DSP**
0	156MHz	362.7k	278.8k	828	256
1	156MHz	345.9k	303.6k	598	31

4.7 Summary

In this chapter, I presented a hardware-software co-design workflow to automatically generate high-performance CNN accelerators. The workflow is able to quantise weights to both fixed-point and shift values at various precisions, and keeps activations to fixed-point numbers. In addition, it transforms batch normalisation to simple affine operations with fixed-point scaling and offset factors. In hardware, the framework utilises the Roll-Unrolled compute pattern and provides flexibility in rolling computations in the channel dimension. As a result, the guided rolling minimises computation while keeping the input stream stall-free. The results showed state-of-the-art performance in terms of model accuracy, latency and throughput. The implemented accelerator for MobileNet is fully pipelined with sub-millisecond latency (0.32ms) and is able to classify at around 3000 frames per second.

Chapter 5

Hardware-aware automated machine learning

The previous two chapters describe several software and hardware techniques for improving the energy efficiency of running DNN inference on different hardware systems. These methods assume pre-trained networks and pre-defined network architectures. In this chapter I will explore opportunities in the model architecture space and the creation of automated tools to aid in their designs. This chapter aims to demonstrate that Automated Machine Learning (AML) is a powerful tool for designing and optimising emerging neural network types and also learning scenarios.

In this chapter, I will first introduce the concepts of Automated Machine Learning (AutoML) and Network Architecture Search (NAS), and discuss why they are particularly useful for designing hardware-aware DNNs. I show two particular applications of NAS in this chapter and demonstrate how they can be an efficient solution for emerging neural network types and new learning setups. First, I demonstrate how state-of-the-art NAS algorithms can be made hardware-aware and show a novel NAS algorithm aiming at finding efficient Graph Neural Networks (GNNs). Later, I show how NAS algorithms can be deployed in a *many-task many-device* setup for few-shot learning.

This chapter includes relevant contents published in:

- Yiren Zhao, Duo Wang, Xitong Gao, Robert Mullins, Pietro Lio, and Mateja Jamnik. Probabilistic dual network architecture search on graphs. In *Deep Learning on Graphs: Method and Applications Workshop for 35th AAAI Conference on Artificial Intelligence (DLG-AAAI 2021), recipient of the best student paper award.*

- Yiren Zhao, Duo Wang, Daniel Bates, Robert Mullins, Mateja Jamnik, and Pietro Lio. Learned low precision graph neural networks. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys 2021), full version is in submission*

- Yiren Zhao, Xitong Gao, Ilia Shumailov, Nicolo Fusi, Robert Mullins. Rapid Model

Architecture Adaption for Meta-Learning. In *submission*

Yiren Zhao conceived the experiments for graph-based NAS methods. Duo Wang later enhanced the search process with a Gumbel-softmax based stochastic method. Yiren Zhao conceived the experiments for the MAML-based NAS and and implemented it as an extension to the MAML++ framework [8]. Ilia Shumailov and Xitong Gao ported other versions using metric-based MAML frameworks as a comparison baseline.

5.1 Motivations for automated machine learning network architecture search

DNNs achieve state-of-the-art results on a wide range of tasks, but tuning the architectures of DNNs on new datasets is getting increasingly difficult. First, the number of possible operations inside a neural network is growing rapidly. In the early years, fully-connected and convolutional layers are the most popular choices for neural networks [93] and only a handful of activation functions are available [178]. The community later witnessed an increasing number of neural network layers, including different normalisations (batched normalisation [83], layer normalisation [11], [160] *etc.*.) and activation functions (SELU [90], ELU [30] *etc.*). The large number of choices for neural network layers makes architectural engineering extremely tedious and it is hard to prove that manually selected operators are anywhere near optimal. I refer to the process of picking individual operators from a set of possible operators as micro-architecture design. Secondly, neural networks are now having complex structures rather than the simple sequential connections found in AlexNet [93] and VGGNets [151]. Later network architecture designs like ResNets [66] and DenseNets [78] start to utilise shortcut connections and have adopted more complex structures. The recently proposed Vision Transformer (ViT) [40] borrows the NLP transformer designs and has sophisticated structures involving multiple attention mechanisms [162].

These two above-mentioned factors together create a large design space for neural network architectures. Most current literature on AutoML and NAS identify the two reasons above as the main motivations for NAS algorithms. In this dissertation, the Network Architecture Search (NAS) are also motivated by the following facts.

First, there are now more and more neural network architecture types. In addition to the rapid growth of the micro-architectures and complex structures of existing DNNs, it is increasingly common to see DNNs applied to new data types such as graph [89] and tabular data [10, 183]. In this chapter, I apply NAS methods to graph neural networks and demonstrate how this automation can help to produce hardware-aware GNNs. The searched GNNs not only have efficient neural network architectures but also have weights and activations being quantised to mixed-precision numbers. I show how NAS is a scalable approach for emerging network types, and how it can be combined with additional search spaces to produce models with hardware awareness. Second, there is now a growing need of applying neural networks to various hardware systems and learning setups. Compared to a few years ago, I now have a diverse set of hardware devices, ranging from server-class GPUs to IoT devices, having the need of executing DNN inference. This chapter then considers a novel *many-task many-device* deployment scenario, in particular, I consider this setup in few-shot learning and demonstrate how NAS can be an effective approach in this setup.

5.2 Network architecture search for GNNs

Graph Neural Networks (GNNs) have been successful in fields such as computational biology [214], social networks [62], knowledge graphs [111], *etc.* The ability of GNNs to apply learned embeddings to unseen nodes or new subgraphs is useful for large-scale ML systems on graph data, ranging from analysing posts on forums to creating product listings for shopping sites [62]. Most large production systems have high throughputs, running millions or billions of GNN inferences per second. This creates an urgent need to minimise the computation cost of GNN inference.

GNN models are typically smaller than CNNs or RNNs, but the need to apply computation at each node or each edge across a graph causes a rapid growth of the computation resources required as the graph size increases. The number of floating point operations (FLOPs) and the maximum possible inference batch size of GNN models are tightly coupled to the size of the input graphs. Figure 2.2 illustrates that a simple 2-layer Graph Attention Network (GAT) [163] can have much higher computational requirements than well-known vision models if input graphs are large. The increased number of FLOPs means a greater inference latency, while a large activation size (the number of temporal activation values generated at run-time) will limit the inference batch size given a limited device RAM budget, and thus decrease the inference throughput.

One common approach to reduce computation and memory costs in neural networks is quantisation [202, 192, 79, 212, 82]. A narrower data format (fewer bits) not only reduces the model size but also introduces the chance of using simpler arithmetic operations on existing or emerging hardware platforms [202, 79]. Quantisation methods to date have focused solely on CNNs and RNNs, but unfortunately cannot be directly adopted in GNNs. First, in CNNs and RNNs, it is the convolutional and fully-connected layers that are quantised [79]. GNNs, however, follow a sample and aggregate approach [205, 62], where a building block of a GNN can be separated into four smaller sub-blocks (Feature Extraction, Attention, Aggregation and Activation); only a subset of these sub-blocks has trainable parameters. Second, GNN blocks involve a design space of attention and aggregation mechanisms where the choice of method influences the required numerical precision. Third, previous CNN NAS methods [174] consider two possible quantisable components (weights and activations) for a single layer. Given n quantisation options, this provides n^2 combinations. I show that a single GNN layer has n^6 quantisation combinations, offering a much larger search space.

In this section, I propose Low Precision Graph NAS (LPGNAS), which aims to automatically design quantised GNNs. The proposed NAS method is single-path, one-shot and gradient-based. Additionally, LPGNAS's quantisation options are at a fine granularity so that different sub-blocks in a graph block can have different quantisation strategies. I describe the details of how to design a hardware-aware GNN NAS method. Section 5.2.1 and Section 5.2.2 describe the architectural and quantisation search spaces of GNNs respectively. I discuss the results of the proposed NAS in Section 5.2.4.

5.2.1 Architectural design space for GNNs

Deep Learning on graphs has emerged into an important field of machine learning in recent years, due to the increase in the amount of graph-structured data. Graph Neural Networks has scored success in a range of different tasks on graph data, such as node classification [163, 62], graph classification [195, 186, 180] and link prediction [194]. There are many variants of graph neural networks proposed to tackle graph structured data. In this work I focus on GNNs applied to node classification tasks based on Message-Passing Neural Networks (MPNN) [53]. The objective of the neural network is to learn an embedding function for node features so that they quickly generalise to other unseen nodes. This inductive nature is needed for large-scale production level machine learning systems since they operate on a constantly changing graph with many incoming nodes (*e.g.* shopping history on Amazon, posts on Reddit *etc.*) [62].

Most of the manually designed GNN architectures proposed for the node classification use message passing neural networks. The list includes, but is not limited to, GCN [89], GAT [163], GraphSage [62], and their variants [149, 193, 22, 167]. These works differ in ways of computing the edge weights, sampling neighbourhood and aggregating neighbour messages. For an emerging type of neural networks, as mentioned previously, neural network architecture search offers a rapid and efficient exploration of the potential architectural design space. Let first look at the architectural design space in details and consider GNNs based on the message-passing mechanism. In each GNN layer, nodes aggregate attention weighted messages from their neighbours and combine these messages with their own feature. Formally, each GNN layer can be described as:

$$h^k = \mathsf{Act}(\mathsf{Aggr}(\mathsf{Atten}(a^k, \mathsf{Linear}(w^k, h^{k-1})))) \tag{5.1}$$

Here $h^{k-1} \in \mathbb{R}^{N \times D}$ is representation for transformed features, where N is the number of nodes in a graph and D is the dimension of the feature. In GNNs, there is typically a linear transformation (linear layer in Figure 5.1) in the form of $W^k h^{k-1}$. a^k are the attention parameters for messages passed from neighbouring nodes (attention in Figure 5.1). AGGR is an aggregation operation for the messages received and is also responsible for combining aggregated messages with features of the current node. Act is a non-linear activation.

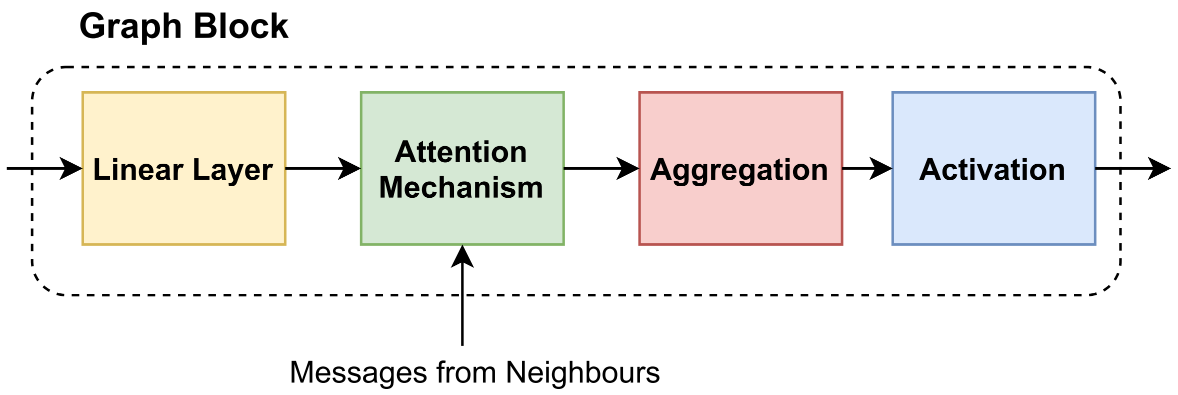


Figure 5.1: The structure of a single graph block consists of four sub-blocks: Linear Layer, Attention, Aggregation and Activation.

A Graph Block is similar to a cell employed in the CNN NAS algorithm DARTS [113]. DARTS uses a weighted sum to combine outputs of all candidate operators. In the proposed NAS, I use the arg max function, which allows only one candidate operator to be active in each training iteration. Let $\bar{o}_{i,k}$ be the k^{th} sub-block in Graph Block i, and $o_{i,k,t}$ be the t^{th} candidate operator for $\bar{o}_{i,k}$. $\bar{o}_{i,k}$ is then computed as:

$$\bar{o}_{i,k} = o_{i,k,t^{max}_{i,k}}, \text{where } t^{max}_{i,k} = \arg\max_{t \in T} P^a_{i,k,t}. \tag{5.2}$$

Here, $P^a_{i,k,t}$ is the probability of the t^{th} candidate operator of sub-block k and layer i assigned by the NAS Controller. P is a probability distribution over all candidate operators, and $P \in \mathbb{R}^{T,K,I}$, where T, K and I are the number of candidate operators, sub-blocks and layers respectively. This hard-max approach considerably reduces memory and computational cost since only one operation is active at any training iterations, whilst still converges, as shown by Wu *et al.* [175] and Xie *et al.* [177]. While the arg max function is non-differentiable, I use a differentiable approximation which ensures that the controller receives learning signals. I will introduce the full details of the proposed NAS algorithm in the next section.

In Table 5.1, I include all the operations described above in our micro-architectural search space. I show all attention types that are in the search space in Table 5.1. The attention types includes a various styles of parametric or non-parametric attention methods. In Table 5.2, I list all activation types that were considered in the architectural search

Table 5.1: Different types of attention mechanisms. W here is parameter vector for attention. $<,>$ is dot product, a_{ij} is attention for message from node j to node i.

Attention Type	Equation
Const	$a_{ij} = 1$
GCN	$a_{ij} = \frac{1}{\sqrt{d_i d_j}}$
GAT	$a_{ij}^{gat} = \mathsf{LeakyReLU}(W_a(h_i \| h_j))$
Sym-GAT	$a_{ij} = a_{ij}^{gat} + a_{ji}^{gat}$
COS	$a_{ij} =< W_{a1}h_i, W_{a2}h_j >$
Linear	$a_{ij} = \mathsf{tanh}(\sum_{j \in N(i)}(W_a h_j))$
Gene-Linear	$a_{ij} = W_g \mathsf{tanh}(W_{a1}h_i + W_{a2}h_j)$

Table 5.2: Different types of activation functions.

Activation	Equation
None	$f(x) = x$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$
Tanh	$f(x) = tanh(x)$
Softplus	$f(x) = \frac{1}{\beta}\log(1 + e^{\beta x})$
ReLU	$f(x) = Max(0, x)$
LeakyReLU	$f(x) = Max(0, x) + \alpha Min(0, x)$
ReLU6	$f(x) = Min(Max(0, x), 6)$
ELU	$f(x) = Max(0, x) + Min(0, \alpha(e^x - 1))$

space. Apart from attention and activation types, I also consider searching for the best hidden feature size, using two fully-connected layers with an intermediate layer with an expansion factor that can be picked from a set $\{1, 2, 4, 8\}$. In terms of aggregation, the choices considered include *mean*, *add* and *max*.

5.2.2 Quantisation search space for graph neural networks

In a single graph block, or a GNN layer, there are four consecutive sub-blocks (Equation (5.3)). Equation (5.3) considers node features h^{k-1} from the $k-1$ layer as inputs, and produces new node features h^k with trainable attention parameters a^k and weights w^k. The four sub-blocks, as illustrated in Equation (5.3), are the Linear block, Attention block, Aggregation block and Activation block; these sub-blocks have operations as search candidates. In Equation (5.4), I label all possible quantisation candidates using Q. The quantisation function Q can be applied not only on learnable parameters (*e.g.*, w^k) but also activation values between consecutive sub-blocks. In addition, I allow quantisation options in Equation (5.4) to be different. For instance, Q_a can learn to have a different quantisation from Q_w, meaning that a single graph block receives a mixed quantisation for different quantisable components annotated in Equation (5.4).

$$h^k = \mathsf{Act}(\mathsf{Aggr}(\mathsf{Atten}(a^k, \mathsf{Linear}(w^k, h^{k-1})))) \tag{5.3}$$

$$\begin{aligned} h^k_{\mathsf{linear}} &= Q_l(\mathsf{Linear}(Q_w(w^k), Q_h(h^{k-1}))) \\ h^k_{\mathsf{atten}} &= Q_{at}(\mathsf{Atten}(Q_a(a^k), h^k_{\mathsf{linear}})) \\ h^k &= \mathsf{Act}(Q_{ag}(\mathsf{Aggr}(h^k_{\mathsf{atten}}))) \end{aligned} \tag{5.4}$$

Table 5.3: Quantisation search space for weights and activations.

Weights			Activations		
Quantisation	Frac Bits	Total Bits	Quantisation	Frac Bits	Total Bits
binary	0	1	fix2.2	2	4
binary	0	1	fix4.4	4	8
ternary	0	2	fix2.2	2	4
ternary	0	2	fix4.4	4	8
ternary	0	2	fix4.8	4	12
fix1.3	3	4	fix4.4	4	8
fix2.2	2	4	fix4.4	4	8
fix1.5	5	6	fix4.4	4	8
fix3.3	3	6	fix4.4	4	8
fix2.4	4	6	fix4.4	4	8
fix4.4	4	8	fix4.4	4	8
fix4.4	4	8	fix4.8	8	12
fix4.4	4	8	fix8.8	8	16
fix4.8	8	12	fix4.8	8	12
fix4.12	12	16	fix4.4	4	8
fix4.12	12	16	fix4.8	8	12
fix4.12	12	16	fix8.8	8	16

The reasons for considering input activation values as quantisation candidates are the following. First, GNNs always consider large input graph data, the computation operates on a per-node or per-edge resolution but requires the entire graph or the entire sampled graph as inputs. The amount of intermediate data produced during the computation is huge and causes a large pressure on the amount of on-chip memory available. Second, quantised input activations with quantised parameters together simplify the arithmetic operations. For instance, $\mathsf{Linear}(Q_w(w^k), Q_h(h^{k-1})))$ considers both quantised h^{k-1} and quantised w^k so that the matrix multiplication with these values can operate on a simpler fixed-point arithmetic.

The quantisation search space identified in Equation (5.4) is different from the search space identified by Wu *et al.* [174] and Guo *et al.* [59]. Most existing NAS methods focusing on quantising CNNs look at quantisation at each convolutional block. In a graph neural network, a single graph block is equivalent to a convolutional block. In a single graph block, I look at more fine-grained quantisation opportunities within the four sub-blocks. The quantisation search considers a wide range of fixed-point quantisations. The weights and activation can pick the numbers of bits in $[1, 2, 4, 6, 8, 12, 16]$ and $[4, 8, 12, 16]$ respectively, and I allow different integer and fractional bits combinations. I list the detailed quantisation options in the Table 5.3. Table 5.3 shows the quantisation search space, each quantisable operation identified will have these quantisation choices available. BINARY means binary quantisation, and TERNARY means two-bit ternary quantisation. FIX$x.y$ means fixed-point quantisation with x integer bits and y bits for fractions. I also demonstrate the number of fractional bits (FRAC BITS) and total number of bits (TOTAL BITS).

In total, as listed in Table 5.3, there are 17 quantisation options; and as illustrated in Equation (5.4), there are six possible components that join the quantisation search, this gives in total $17^6 = 24137569$ quantisation combinations for a Graph Neural Network.

5.2.3 Low precision graph neural network architecture search

Algorithm 2 LPGNAS algorithm

```
1: Inputs: $x$, $y$, $x_v$, $y_v$, $M$, $M_a$, $M_q$, $K$, $\alpha$, $\beta$
2: $\mathsf{Init}(w, w_a, w_q)$
3: for $i = 0$ to $M - 1$ do
4:     $N$ = NoiseGen($i$, $\alpha$)
5:     $P_a, P_q = g_a(w_a, x_{\mathsf{val}}, N), g_q(w_q, x_{\mathsf{val}}, N)$
6:     $\pi_a, \pi_q = \arg\max(P_a), \arg\max(P_q)$
7:     for $i = 0$ to $K - 1$ do
8:         $\mathcal{L} = \mathsf{Loss}(x, y, \pi_a, \pi_q)$
9:         $w = \mathsf{Opt}_w(\mathcal{L})$
10:    end for
11:    $\mathcal{L}_v = \mathsf{Loss}(x_v, y_v, \pi_a, \pi_q)$
12:    if $e > M_a$ then
13:        $w_a = \mathsf{Opt}_{w_a}(\mathcal{L}_v)$
14:    end if
15:    if $e > M_q$ then
16:        $\mathcal{L}_q = \mathsf{QLoss}(P_a, P_q)$
17:        $w_q = \mathsf{Opt}_{w_q}(\mathcal{L}_v + \beta L_q)$
18:    end if
19: end for
```

I describe the Low Precision Graph Network Architecture Search (LPGNAS) algorithm in Algorithm 2. (x, y), (x_v, x_v) are the training and validation data respectively. M is the total number of search epochs, and M_a and M_q are the epochs to start architectural and quantisation search. Before the number of epochs reaches M_a or M_q, LPGNAS randomly picks up samples and warms up the trainable parameters in the supernet. K is the number of steps to iterate in training the supernet. After generating noise N, I use this noise in architectural controller g_a and quantisation controller g_q together with the validation data x_v to determine the architectural π_a and quantisation π_q choices. After reaching the pre-defined starting epoch (M_a or M_q), LPGNAS starts to optimise the controllers' weights (w_a and w_q). I choose the hyper-parameters $M_q = 20, M_a = 50, \alpha = 1.0, \beta = 0.1$, unless specified otherwise. I provide an ablation study in the Appendix to justify our choices of hyper-parameters. Notice that QLoss estimates the current hardware cost based on both architectural and quantisation probabilities. As shown in Equation (5.5), I define S as a set of values that represents the costs (number of parameters) of all possible architectural options: for each value s in this set, I multiply it with the probability p_a from architecture probabilities P_a generated by the NAS architecture controller. Similarly, I produce the weighted-sum for quantisation from the set of values S_q that represents costs of all possible quantisations. The dot product $<,>$ between these two weighted-sum vectors produces the quantisation loss L_q.

$$\begin{aligned} L_q &= \mathsf{QLoss}(P_a, P_q) \\ &=< \sum_{s \in S, p_a \in P_a} (s * p_a), \sum_{s_q \in S_q, p_q \in P_q} (s_q * p_q) > \end{aligned} \tag{5.5}$$

Figure 5.2 is an illustration of the LPGNAS algorithm. I use two controllers (g_a and g_q) for architecture and quantisation (named as Controller and Q-Controller in the diagram) respectively. The controllers use trainable embeddings connected to linear layers to produce architecture and quantisation probabilities. The architecture controller provides probabilities (P_a) for picking architectural options of graph blocks and also the router. The router is in charge of determining how each graph block shortcuts to the subsequent graph blocks [205]. In addition, the quantisation controller provides quantisation probabilities (P_q) to both modules in graph blocks and the linear layers in the router. In a graph block, only the linear and attention layers have quantisable parameters and have matrix multiplication operations; neither activation nor aggregation layers have any trainable or quantisable parameters. However, all input activation values of each layer in the graph block are quantised. The amount of intermediate data produced during the computation causes memory pressure on many existing or emerging hardware devices, so even for modules that do not have quantisable parameters, I choose to quantise their input activation values to help reduce the amount of buffer space required.

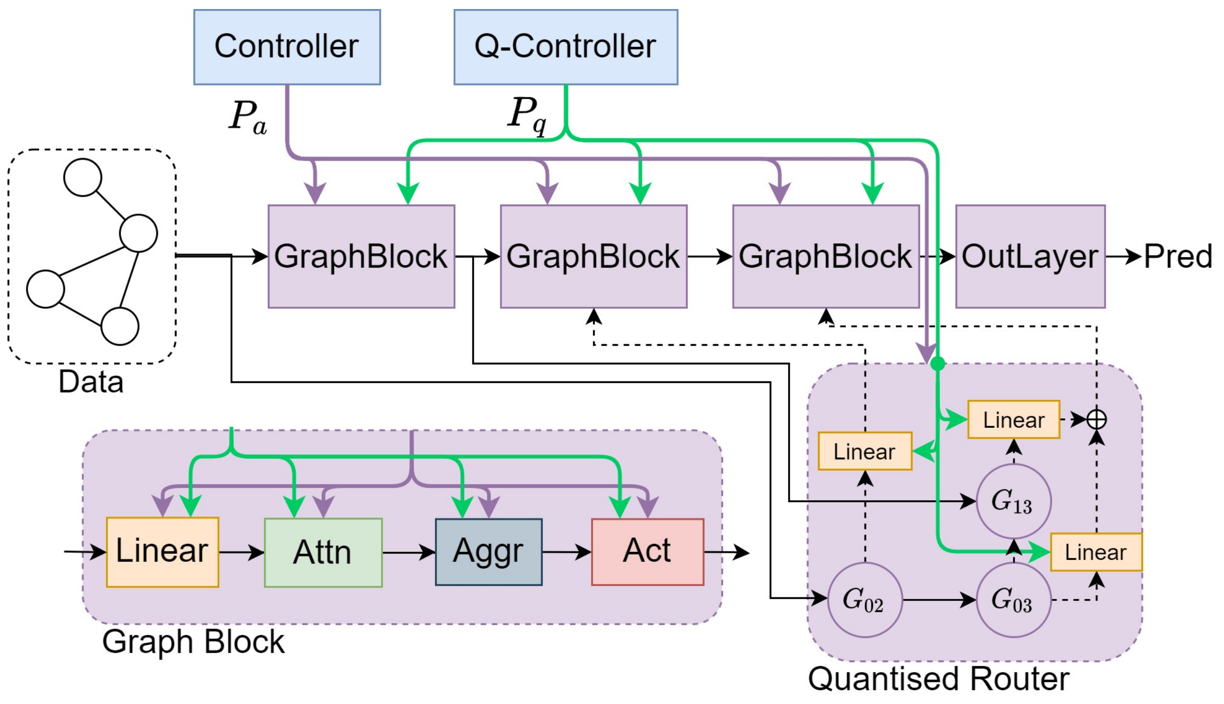


Figure 5.2: An overview of the LPGNAS architecture. P_a denotes controller output (Purple) for selecting different operations within each graph block, and for routing shortcut connections between graph blocks. P_q denotes quantisation controller (Q-Controller) output (Green) for selecting quantisations for operations within graph blocks and shortcut connections. G_{ij} are gating functions conditioned on P_a. Solid lines are input streams into the router while dashed lines are output streams.

5.2.4 Evaluating low precision graph network architecture search (LPGNAS)

In Section 5.2.4.1, I present the results of running LPGNAS on the Citation datasets, these datasets take the entire input graph as a single input to the neural networks. Section 5.2.4.2 shows the results of LPGNAS on graph sampling based learning, there exists a sampler on the input graphs and the sampled subgraphs are then the inputs to the neural network. The graph sampling approach is a common technique to deal with large input graphs.

5.2.4.1 LPGNAS on Citation datasets

The Citation datasets include nodes representing documents and edges representing citation links. The task is to distinguish which research field the document belongs to [184].

Table 5.4 shows the performance of GraphSage [62], GAT [163], JKNet [179], PDNAS [205] and LPGNAS on the citation datasets with a partition of 0.6, 0.2, 0.2 for training, validation and testing respectively. For the quantisation options of GraphSage, GAT and JKNet, I manually perform a grid search on the Cora dataset. The grid search considers various weight and activation quantisations in a decreasing order. I reimplemented

Table 5.4: Accuracy and size comparison on Cora, Pubmed and Citeseer with a data split of 60%, 20% and 20% for training, validation and testing. Our results are averaged across 3 independent runs.

		Cora				CiteSeer				PubMed			
Method	Quan	Acc %	Size KB	Act Size MB	BitOPs G	Acc %	Size KB	Act Size MB	BitOPs G	Acc %	Size KB	Act Size MB	BitOPs G
GraphSage	float	74.5 ± 0.0	92.3	62.1	48.3	75.3 ± 0.0	237.5	197.1	58.6	85.3 ± 0.1	32.2	157.7	349.1
GraphSage	w10a12	74.3 ± 0.1	28.8	23.3	5.4	75.1 ± 0.1	74.2	73.9	6.6	85.0 ± 0.0	10.1	59.1	39.0
GAT	float	88.9 ± 0.0	369.5	62.1	25.1	75.9 ± 0.0	950.3	197.1	30.2	86.1 ± 0.0	129.6	157.7	181.9
GAT	w4a8	88.8 ± 0.1	46.2	15.5	**0.73**	68.0 ± 0.1	118.8	49.3	**0.89**	82.0 ± 0.0	16.2	39.4	**5.3**
JKNet	float	88.7 ± 0.0	214.9	139.7	1018.5	75.5 ± 0.0	505.2	443.5	3230.1	87.6 ± 0.0	94.5	354.9	2588.2
JKNet	w6a8	88.7 ± 0.1	**40.3**	26.2	23.9	73.2 ± 0.1	94.7	83.2	75.7	$86.1 \pm 0.1\%$	17.7	66.5	60.7
PDNAS-2	float	89.3 ± 0.1	192.2	62.1	50.4	$\mathbf{76.3 \pm 0.3}$	478.6	197.1	62.3	89.1 ± 0.2	72.8	157.7	77.3
PDNAS-3	float	89.3 ± 0.1	200.0	93.2	83.2	75.5 ± 0.3	494.4	295.6	77.3	89.1 ± 0.2	81.4	236.6	98.4
PDNAS-4	float	89.8 ± 0.3	205.0	124.2	102.3	75.6 ± 0.2	500.0	404.5	130.2	89.2 ± 0.1	102.7	340.3	244.3
PDNAS-4	w8a8	86.9 ± 0.1	51.3	31.1	12.5	69.3 ± 0.1	125.0	101.1	32.1	88.9 ± 0.1	25.7	85.1	71.2
PDNAS-4	w12a16	88.8 ± 0.2	76.9	46.7	27.3	74.4 ± 0.2	187.5	151.1	44.2	89.0 ± 0.1	38.5	126.7	92.3
LPGNAS	mixed	$\mathbf{89.8 \pm 0.0}$	67.3	**10.5**	9.3	76.2 ± 0.1	**56.5**	**14.6**	19.9	$\mathbf{89.6 \pm 0.1}$	45.6	**29.7**	20.8

GraphSage [62], GAT [163] and JKNet [179] for quantisation, and provide the architecture choices. For the manual baseline networks, I used the quantisation strategy found on Cora for Citeseer and Pubmed. For LPGNAS, I searched with a set of base configurations that I clarified in the Appendix.

There are several hyperparameters in the LPGNAS algorithm (Algorithm 2). As mentioned, I pick $N_q = 20, N_a = 50, \alpha = 1.0, \beta = 0.1, \mathsf{lr} = 0.005$ through a hyperparameter study. For β, N_q and N_a, I justify the choices in the graph below by sweeping across different values of β, N_a and N_q on the Pubmed dataset (Figure 5.3).

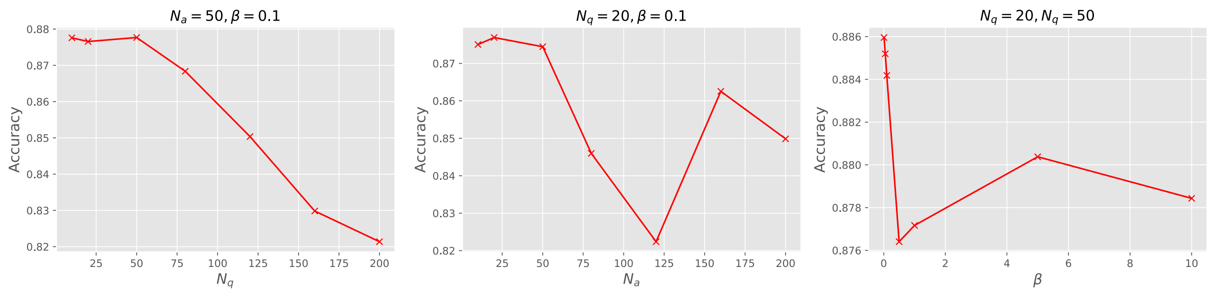


Figure 5.3: Collected statistical information for quantisation, the horizontal axis shows the chosen bit-width and the vertical axis shows the occurences.

The results in Table 5.4 suggest that LPGNAS shows better accuracy on both Cora and Pubmed with quantised networks. In addition, although LPGNAS does not show the smallest sizes on these two datasets, it is only slightly bigger than the manual baselines but shows a much better accuracy. On Citeseer, LPGNAS only shows slightly worse accuracy (around 0.1% less) with a considerably smaller size (around 9× reduction in model sizes).

Table 5.5: Accuracy and size comparison on Amazon-Computers and Amazon-Photo [146] Our results are averaged across 3 independent runs.

Method	Quan	Amazon-Computers				Amazon-Photos			
		Acc %	Size KB	Act Size MB	Bitops G	Acc %	Size KB	Act Size MB	Bitops G
SageNet	float	84.0 ± 0.0	49.8	4.6	6.8	84.0 ± 0.1	49.8	4.6	6.8
SageNet	w4a8	83.9 ± 0.1	6.2	1.1	0.21	83.9 ± 0.1	6.2	1.1	0.21
GAT	float	84.4 ± 0.3	199.8	4.6	3.6	84.2 ± 0.1	199.8	4.6	3.6
GAT	w4a8	81.8 ± 1.5	25.0	1.1	**0.10**	83.5 ± 0.5	25.0	1.1	**0.10**
JKNet	float	87.6 ± 0.0	130.5	10.1	74.5	87.7 ± 0.1	130.5	10.1	74.5
JKNet	w4a8	38.0 ± 0.0	16.3	2.5	2.3	38.0 ± 0.0	16.3	2.5	2.3
PDNAS-4	w4a8	85.6 ± 0.1	100.0	2.7	2.1	84.6 ± 0.1	75.4	1.4	3.3
LPGNAS	mixed	$\mathbf{90.5 \pm 0.0}$	157.3	3.7	3.5	$\mathbf{89.7 \pm 0.0}$	164.0	3.7	4.5
LPGNAS-small	mixed	88.7 ± 0.1	**3.5**	**1.0**	1.3	88.6 ± 0.1	**3.6**	**0.81**	1.2

Table 5.6: Accuracy and size comparison on Flickr [189], Cora-full [13] and Yelp [189]. Our results are averaged across 3 independent runs.

Method	Quan	Flickr				CoraFull				Yelp			
		Acc %	Size KB	Act Size MB	Bitops G	Acc %	Size KB	Act Size MB	Bitops G	Micro F1	Size KB	Act Size MB	Bitops G
SageNet	float	49.9 ± 0.0	32.5	2.9	6.2	66.4 ± 0.1	562.3	39.4	13.4	23.6 ± 0.0	26.1	1.6	19.1
QSageNet	w4a8	45.0 ± 0.0	**4.1**	0.72	**0.20**	55.7 ± 0.2	70.3	12.3	0.41	24.8 ± 0.0	3.3	0.39	0.59
GAT	float	42.4 ± 0.1	130.6	2.9	3.3	62.0 ± 0.1	2249.3	49.4	6.8	24.6 ± 0.5	104.4	1.6	9.7
QGAT	w4a8	42.4 ± 0.0	16.3	0.73	0.098	53.8 ± 0.2	281.2	12.3	0.21	14.2 ± 0.0	13.0	**0.39**	0.30
JKNet	float	50.6 ± 0.0	95.5	6.6	48.5	69.0 ± 0.1	1162.8	112.3	809.9	32.6 ± 0.3	94.1	3.5	27.6
QJKNet	w4a8	42.3 ± 0.0	11.9	1.6	1.5	4.4 ± 0.0	145.3	28.1	25.3	31.0 ± 2.2	11.8	0.87	0.86
PDNAS-4	w4a8	68.6 ± 0.1	223.5	8.3	12.4	72.6 ± 0.2	657.1	11.1	27.2	$\mathbf{42.4 \pm 0.0}$	62.0	1.2	1.4
LPGNAS	mixed	$\mathbf{74.6 \pm 0.2}$	113.8	2.8	1.6	$\mathbf{77.7 \pm 0.0}$	573.3	7.4	25.8	40.5 ± 0.1	6.4	3.2	1.2
LPGNAS-small	mixed	69.8 ± 0.4	6.4	**0.31**	0.21	69.8 ± 0.2	141.5	**3.2**	**0.16**	30.9 ± 0.1	**1.5**	1.8	**0.14**

5.2.4.2 LPGNAS for graph sampling based learning

Table 5.5 and Table 5.6 show how LPGNAS performs on large datasets sampled using the recently proposed GraphSaint sampling [189]. The detailed configuration of the sampler is in the Appendix. I provide additional baselines on these datasets since the original GraphSage [62], GAT [163], JKNet [179] under-fit this challenging task. I thus implemented V2 versions of the baselines that have larger channel counts; the architecture details are reported in the Appendix. In addition, for quantising the baseline models I uniformly applied a w4a8 (4-bit weights, 8-bit input activations) quantisation. Although I train on a sampled sub-graph, evaluating networks requires a full traversal of the complete graph. Since the datasets considered are large, traversing the entire graph adds a huge computation overhead. If I use the previous grid-search for all quantisation possibilities (roughly 16^4 in our case) for the all baseline networks, this will require a huge amount of search time. So I fixed the quantisation strategy to w4a8 for the baseline networks.

The results in Table 5.5 and Table 5.6 suggest that LPGNAS found the models with

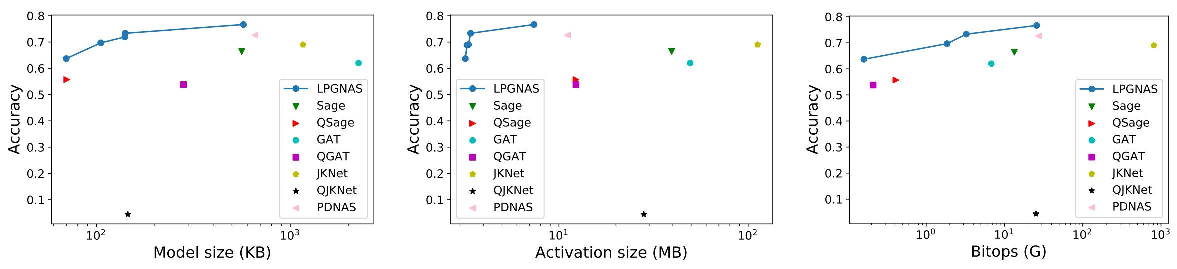


Figure 5.4: Pareto frontier of LPGNAS on Coraf-full with baseline models. I show the trade-off between model sizes, activation sizes and bitops, LPGNAS demonstrates a Pareto dominance.

best accuracy and also with a relatively competitive model size when compared to a wide range of manual baselines. To further demonstrate that LPGNAS is a flexible NAS method, I adjust the base configurations to provide models that are equal or smaller than the smallest baseline networks in Table 5.5 and Table 5.6 and name them LPGNAS-small. I describe how I turn the knobs in LPGNAS by adjusting the base configurations in the Appendix. It is clear, in both Table 5.5 and Table 5.6, that LPGNAS outperforms all baselines in terms of accuracy or micro-F1 scores. LPGNAS-small, on the other hand, trades-off the accuracy for a better model size, however it shows better performance than baseline models that have a similar size budget. I also visualise the performance of LPGNAS on sampled datasets using the Pareto frontiers plot for Cora-full in Appendix. As expected, LPGNAS shows a Pareto dominance compared other manually designed networks.

LPGNAS is able to achieve significant performance gains in terms of accuracy over the baseline networks. For instance, on Flickr, CoraFull, and Yelp, I show on average an around 20% increase compared to even the full-precision baseline models, while LPGNAS produced quantised models. The performance gap between quantised baselines and LPGNAS is even greater. On the other hand, LPGNAS is able to produce extremely small models. For instance, LPGNAS-Small is able to produce models that are around 50× smaller on Amazon-Computers and Amazon-Photos while only suffer from an around 1% drop in accuracy compared to standard LPGNAS.

To further prove the point that LPGNAS generates more efficient networks, I sweep across different configurations and different quantisation regularisations to produce Figure 5.4. LPGNAS shows a Pareto dominance compared to all evaluated methods, meaning that it strikes the best combination between computational complexity and accuracy. Figure 5.4 shows that models can run with larger batch sizes (reduction in activation size), and also run with less computation (reduction in bitops).

Table 5.7: Search cost in GPU hours, all experiments are conducted on an NVIDIA GeForce RTX 2080 Ti GPU.

Dataset	Cora	Citeseer	Pubmed	Cora-full	Flickr	Yelp	Amazon-photos	Amazon-computers
LPGNAS	3.2	3.6	4.2	11.8	11.0	11.8	11.4	11.2
JKNet-32	6518.5	8165.6	5289.1	2180.6	3062.1	9116.7	1739.8	1786.2

5.2.5 Quantisation search time

One advantage of using Network Architecture Search is the reduction of search time. Previous NAS methods on GNNs demonstrated the effectiveness of their methods by reducing the amount of search time required to find the best network architecture [205, 51, 209]. In this work, I not only reduced the amount of manual tuning time required for network architectures but also shortened the amount of time required to optimise the numerical precision of each sub-block.

To further illustrate that LPGNAS has significantly reduced the amount of search time required, I provide Table 5.7 to show how LPGNAS compares to a fixed JKNet-32 architecture in terms of the amount of GPU hours spent for searching for the best quantisation options. Because of the limited resources I have, I estimate the quantisation search cost of a JKNet-32 by running 5 different randomly selected quantisation combinations and multiply the averaged training time with the total number of quantisation options.

Table 5.7 shows that LPGNAS can significantly reduce the search time. For instance, on Citeseer, the search time can be reduced by around 2270×. It is worth noting that, when performing this quantisation search time comparison, I do not consider the architectural search space of the baseline, meaning that the baseline (JKNet) is a fixed-architecture. LPGNAS also searched for architecture choices, which would further increases the search time of the baseline if they are considered.

5.2.6 Quantisation statistics and limitations

In order to fully understand the properties of different quantisable sub-blocks in GNNs, I report the searched quantisation strategies on more than 100 search runs across 8 different datasets (Cora, Pubmed, Citeseer, Amazon-Computers, Amazon-Photos, Flickr, CoraFull and Yelp). The idea is to reveal some common quantisation properties or trends using LPGNAS on different datasets. I collect the quantisation decisions made by LPGNAS for both weights and activations and present them in Figure 5.5. The possible quantisation levels for weights and activations are [1, 2, 4, 6, 8, 12, 16] and [4, 8, 12, 16] bits respectively.

For weights, we see that narrow ternary or even binary values are often sufficient for the hidden and attention sub-blocks, however, shortcut connections prefer a larger bit-width (around 4 bits) for weights. In contrast, quantising activations show in general a trend of using larger bit-widths (around 8 bits). Based on these results, if one would like to manually quantise GNNs, I would suggest to start with around 4 bits for weights and 8 bits for activations. However, it is worth mentioning that LPGNAS is able to search for the best combination of quantisation and architectural decisions. For instance, I observed LPGNAS was able to choose operations with less parameters but use higher bit-widths. Nevertheless, the large number of runs across a relatively wide range of datasets aim at sharing empirical insights for people that are manually tuning quantisations for GNNs. For GNN services that are whether energy, memory or latency critical, and for future AI ASIC designers focusing on GNN inference, the gathered quantisation statistics can be a useful guideline for fitting a suitable quantisation method to their networks.

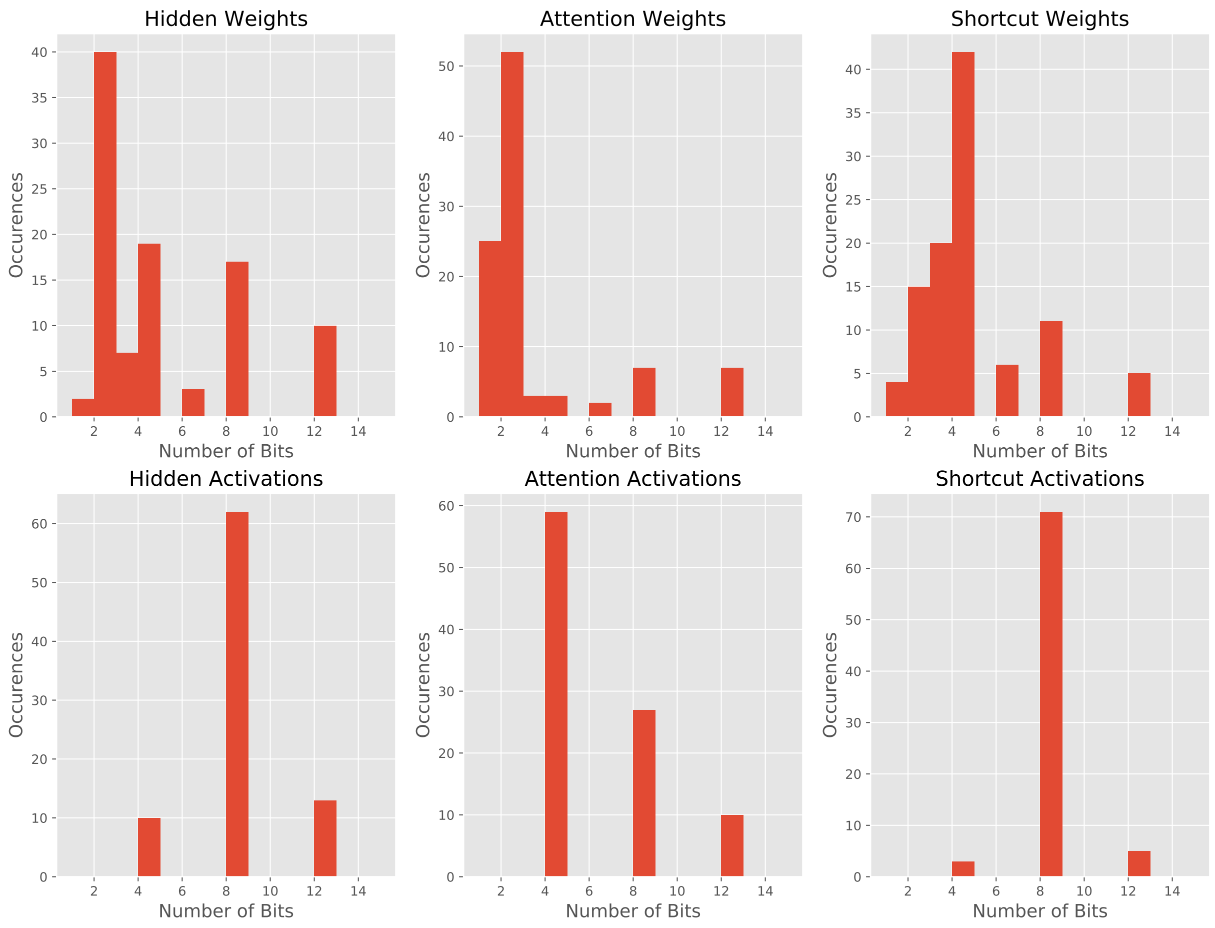


Figure 5.5: Collected statistical information for quantisation on Cora, Citeseer, PubMed, Amazon-Computers, Amazon-Photos, Flickr, CoraFull and Yelp, the horizontal axis shows the chosen bit-width and the vertical axis shows the occurences of quantisations (not weighted by the number of weights). A bit-width count of 16 is not present because it has never been selected by LPGNAS.

It is also interesting to observe that in both Table 5.5 and Table 5.6, some of the manually designed GNNs suffer from a large accuracy drop when applying a 4-bit weight and

8-bit activation fixed-point quantisation (w4a8). For instance, both quantised JKNet and JKNet-V2 drop down to 4.4% accuracy on the Cora-full dataset (Table 5.5). However, the collected statistics of LPGNAS suggest that w4a8 is sufficient, proving that modifications of network architectures are the key element in order for networks to maintain accuracy with aggressive quantisation strategies.

The applied fixed-point quantisation is a straight-forward one, many researchers have proposed more advanced quantisation schemes on CNNs or RNNs [202, 192, 150]. I suggest future research of investigating how these quantisation strategies work on GNNs should consider w4a8 fixed-point quantisation as a baseline case. Because the collected statistics suggest that w4a8 fixed-point quantisation is the most aggressive quantisation for searched GNNs while guaranteeing minimal loss of accuracy, and manual GNNs often do not perform well using the w4a8 setup.

5.3 Network architecture search in a multi-task multi-device few-shot learning setup

Existing Network Architecture Search (NAS) methods show promising performance on image [215, 113], language [58, 152] and, as I have shown in the previous section, graph data [205]. This automation not only reduces the amount of human effort required for architecture tuning but also produces architectures with state-of-the-art performance in domains like image classification [215] and language modeling [152] and graph node classification (Section 5.2). Most NAS methods today focus on a single task while targeting a single hardware platform, *e.g.* a mobile phone or a data center class GPU. Real-life model deployments with multiple tasks and various hardware platforms will significantly prolong this process. As illustrated in Figure 5.6, a common design flow is to re-engineer the architecture and train for different task(T)-hardware(H) pairs with different constraints (C). The architectural engineering phase can be accomplished whether manually or by using an established NAS procedure. The major challenge remains to be how to design efficient algorithmic method to overcome the increased $\mathcal{O}(THC)$ search complexity described in Figure 5.6.

Few-shot learning systems follow exactly this *many-task many-device* setup, since it considers deployments on different user devices on key applications such as facial [54] and speech recognition [74]. A task in few-shot learning normally takes a N-way K-shot formulation, where it contains N classes with K support samples and Q query samples in each class. Model-Agnostic Meta-Learning (MAML), incorporating the idea of learning to learn, builds a meta-model using a great number of training tasks, and then adapts the meta-model to unseen test tasks using only a very small number of gradient updates [47]. MAML then becomes a powerful and elegant approach for few-shot learning, its ability to quickly adapt to new tasks can potentially shrink the $\mathcal{O}(THC)$ complexity illustrated in Figure 5.6 to $\mathcal{O}(HC)$. In the meantime, hardware-aware NAS methods [18, 17, 181], *e.g.* the train-once-for-all technique [18], support deployments of searched models to fit to different hardware platforms with various latency constraints. These hardware-aware NAS techniques reduce the search complexity from $\mathcal{O}(THC)$ to $\mathcal{O}(T)$ [17].

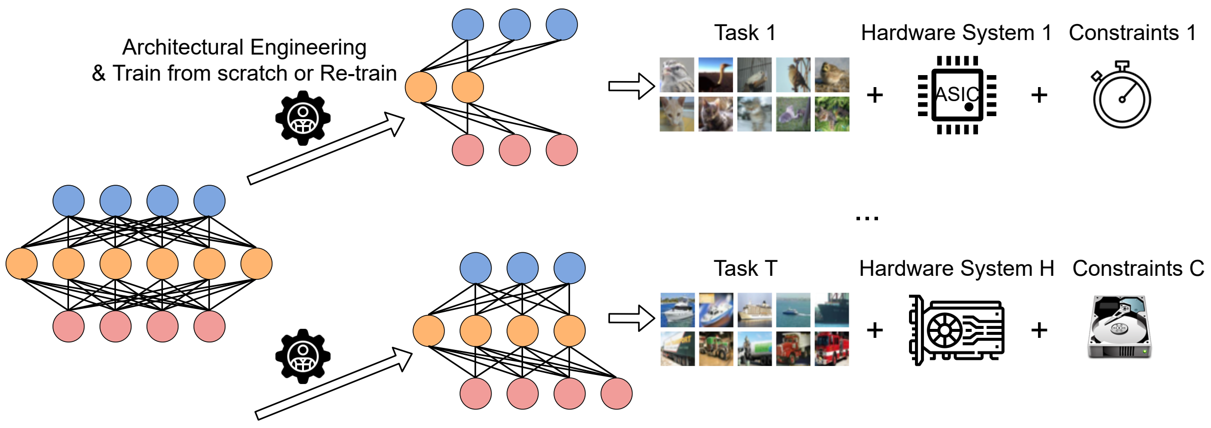


Figure 5.6: Deploying networks in a *many-task many-device* setup. This implies a large search complexity $\mathcal{O}(THC)$ and requires a huge amount of architectural engineering time.

I propose a novel Hardware-aware Meta Network Architecture Search (H-Meta-NAS). Integration of the MAML framework into hardware-aware NAS would theoretically simplify the search complexity from $\mathcal{O}(THC)$ to $\mathcal{O}(1)$, allowing a rapid adaption of model architectures to unseen tasks on new hardware systems. However, I identified the following challenges to this integration:

- The classic NAS search space contains many over-parameterised sub-models, this makes it hard to tackle the over-fitting phenomenon in few-shot learning.
- Hardware-aware NAS profiles latency for sub-networks on each task-hardware pair, this profiling can be prolonged significantly with a great number of tasks and, more importantly, if the targeting device has scarce computation resources.

To tackle these challenges, I propose to use Global Expansion (GE) and Adaptive Number of Layers (ANL) to allow a drastic change in model capabilities for tasks with varying difficulties. The experiments later proved that such changes alleviate over-fitting in few-shot learning and improve the accuracy significantly. I also present a novel layer-wise profiling strategy to allow reuse of profiling information across different tasks.

5.3.1 Few-shot learning in the MAML framework

Inspired by the ability for humans to learn from only a few tasks and generalise the knowledge to unseen problems, a meta learner is trained over a distribution of tasks with the hope of generalising its knowledge to new tasks [47].

$$\arg\min_{\theta}(\mathbb{E}_{\mathcal{T}\in\mathbb{T}}[\mathcal{L}_\theta(\mathcal{T})]) \tag{5.6}$$

Equation (5.6) captures the optimisation objective of meta-learning, where optimal parameters are obtained through optimising on a set of *meta-training* tasks. Current mainstream approaches that use meta-learning to tackle few-shot learning problems can be roughly categorised into three major types: Memory-based, Metric-based and Optimisation-based.

Memory-based method utilises a memory-augmented neural network [123, 52] to memorise meta-knowledge for a fast adaption to new tasks. Metric-based methods aim to meta-learn a high-dimensional feature representation of samples, and then apply certain metrics to distinguish them. For instance, Meta-Baseline utilises the cosine nearest-centroid metric [26] and DeepEMD applies the Earth Mover's Distance [191]. Optimisation-based method, on the other hand, focuses on learning a good parameter initialisation (also known as *meta-parameters* or *meta-weights*) from a great number of training tasks, such that these meta-parameters adapt to new few-shot tasks within a few gradient updates. The most well-established Optimisation-based method is Model-Agnostic Meta-Learning (MAML) [47]. MAML is a powerful yet simple method to tackle the few-shot learning problem, since its adaption relies solely on gradient updates. Antoniou *et al.* later demonstrate MAML++, a series of modifications that improved MAML's performance and stability [8].

While the meta-learning framework is increasingly used to solve few-shot learning challenges, little attention has been paid to the run-time efficiency of these approaches. Meta-learning has been explored in key applications such as facial and speech recognition [74, 54] for mobile deices. Real-life deployments on these devices resemble a *many-task many-device* scenario, where learning on each user's data is a few-shot learning task and different hardware platforms represent different types of under-deployment devices. Memory-based and Metric-based meta-learning methods are then challenged by the hardware or latency constraints: Memory-based methods need additional storage space (at least double) and Metric-based approaches use multiple inference runs (at least two) for a single image classification. Our proposed NAS method then follows the simple yet effective MAML++ framework. After adapting the model to new tasks, MAML++ executes exactly one network inference run for a single test sample without additional memory usage.

Several NAS methods are proposed under the MAML framework [88, 144, 107], these methods successfully reduce the search complexity from $\mathcal{O}(T)$ to $\mathcal{O}(1)$. However, some of these methods do not show significant performance improvements compared to carefully designed MAML methods (*e.g.* MAML++) [88, 144]. In the meantime, some of these MAML-based NAS methods follow the Gradient-based approach and operate on complicated cell-based structures [107]. I illustrate later how cell-based NAS causes an undesirable effect on latency, and also meets fundamental scalability challenges when trying to deploy in a *many-task many-device* setup.

5.3.2 Hardware-aware network architecture search for meta-learning

Figure 5.7 is an overview of the proposed H-Meta-NAS flow. There are three stages in the search process. First, I meta-train a super-net. The super-net contains many possible sub-networks (roughly 10^9 sub-networks using the VGG9 backbone) (Section 5.3.2.3). I use a layer-wise profiling to construct a hash-table for different hardware systems, there is no training involved in the profiling stage, the details of the profiling is discussed in Section 5.3.2.4. The third phase is to adapt not only the meta-parameters (θ) but also the meta-architecture (α) from the super-net to a particular task (Section 5.3.2.5) with specific hardware constraints on a specific hardware system.

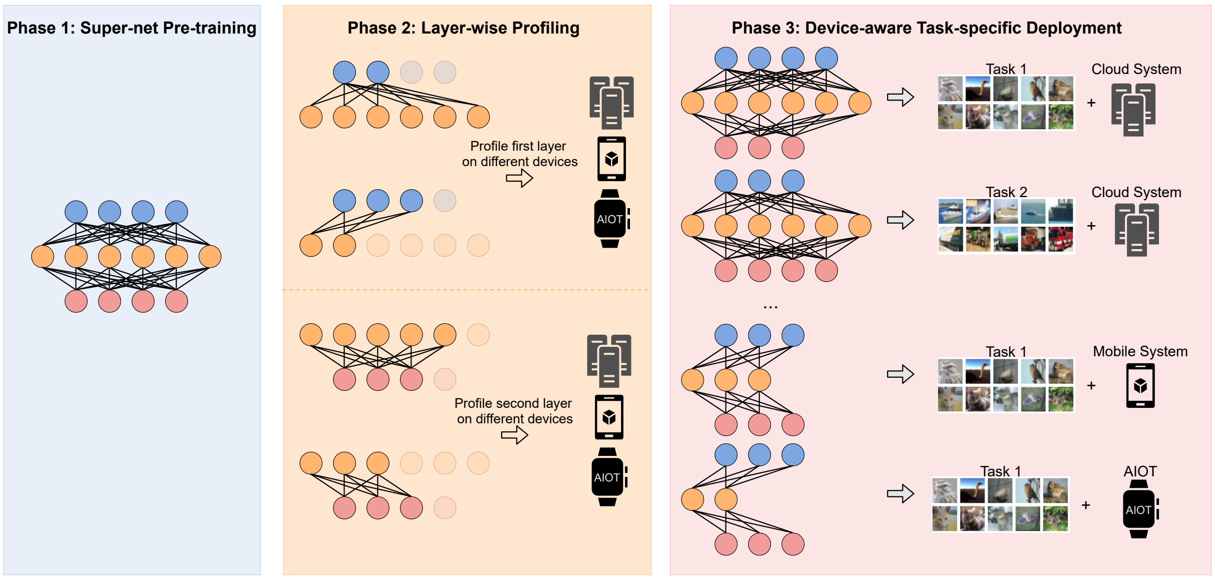

Phase 1: Super-net Pre-training
Phase 2: Layer-wise Profiling
Profile first layer on different devices
AIOT
Profile second layer on different devices
AIOT
Phase 3: Device-aware Task-specific Deployment
Task 1
Cloud System
Task 2
Cloud System
...
Task 1
Mobile System
Task 1
AIOT
AIOT

Figure 5.7: An overview of the three main stages of the proposed H-Meta-NAS algorithm. The first stage is to meta-train a super-net, the second stage is to adapt the super-net to a particular task, and the third stage is to adapt the super-net to a particular hardware system.

5.3.2.1 Problem formulation

In the MAML setup, I consider a set of tasks $\mathbb{T}$ and each task $\mathcal{T}_i \in \mathbb{T}$ contains a support set $\mathcal{D}_i^s$ and a query set $\mathcal{D}_i^q$. The support set is used for task-level learning while the query set is in charge of evaluating the meta-model. All tasks are divided into three sets, namely meta-training ($\mathbb{T}_{train}$), meta-validation ($\mathbb{T}_{val}$) and meta-testing ($\mathbb{T}_{test}$) sets.

Equation (5.7) formally states the objective of the pre-training stage illustrated in Figure 5.7 Phase 1. The objective of this process is to optimise the parameters θ of the super-net for various sub-networks sampled from the architecture set $\mathbb{A}$. This will ensure the proposed H-Meta-NAS to have both the meta-parameters and meta-architectures ready for the adaption to new tasks. Section 5.3.2.3 describes the details of the meta-training and a progressive shrinking strategy to prevent sub-model interference.

$$\arg\min_{\theta} \mathbb{E}_{\alpha\sim p(\mathbb{A})}[\mathbb{E}_{\mathcal{T}\in\mathbb{T}_{train}}[\mathcal{L}_{\theta}(\mathcal{T},\alpha)]] \tag{5.7}$$

Equation (5.8) describes how H-Meta-NAS adapts network architectures to a particular task $\mathcal{T}$ with a given hardware constraint C_h (Phase 3 in Figure 5.7). In practice, using the support set data $\mathcal{D}_i^s$ from a target task $\mathcal{T}_i$, I apply a genetic algorithm for finding the optimal architectures α^*. I discuss further how this process is designed in Section 5.3.2.5.

$$\begin{aligned} \alpha^* = \min_{\alpha} \sum_{\alpha\in\mathbb{A}} \mathcal{L}_{\theta}(\mathcal{D}_i^s, \alpha) \\ \text{s.t.} \quad \mathcal{C}(\alpha) \leq C_h \end{aligned} \tag{5.8}$$

5.3.2.2 Architecture space

H-Meta-NAS considers a search space composed of different kernel sizes, number of channels and activation types. I mostly consider a VGG9-based NAS backbone, that is a 5-layer CNN model with the last layer being a fully connected layer. I chose this NAS backbone because both MAML [47] and MAML++ [8] used a VGG9 model architecture. The details of this backbone are in Appendix.

I allow kernel sizes to be picked from $\{1, 3, 5\}$, channels to be expanded with a set of scaling factors $\{0.25, 0.5, 0.75, 1, 1.5, 2, 2.25\}$ and also six different activation functions (details in Appendix). For a single layer, there is $3 \times 7 \times 6 = 126$ search options. H-Meta-NAS also contains an Adaptive Number of Layers strategy, the network is allowed to use a subset of the total layers in the supernet with a maximum usage of 4 layers. The whole VGG9-based backbone then gives us in total $126^4 \times 4 \approx 10^9$ possible neural network architectures.

In addition, to demonstrate the ability of H-Meta-NAS on a more complex NAS backbone. I also studied an alternative ResNet12-based NAS backbone, that has approximately 2×10^{24} possible sub-networks.

5.3.2.3 Pre-training strategy

As illustrated by prior work [18], progressively shrinking the super-net during meta-training can reduce the sub-model interference, I observe the same phenomenon and then use a similar progressive shrinking strategy in H-Meta-NAS, the architectural sampling process

$\alpha \sim p(\mathbb{A})$ will pick the largest network with a probability of p, and randomly pick other sub-networks with a probability of $1-p$. I apply an exponentially decay strategy to p:

$$p = p_e + (p_i - p_e) \times exp(-\alpha \times \frac{e - e_s}{e_m - e_s})) \tag{5.9}$$

p_e and p_i are the end and initial probabilities. e is the current number of epochs, and e_s and e_m are the starting and ending epochs of applying this decaying process. α determines how fast the decay is. In our experiment, I choose $p_i = 1.0$ and $e_s = 30$, because the super-net reaches a relatively stable training accuracy at that point. I then start the decaying process, and the value $\alpha = 5$ is determined through the following hyper-parameter study shown in Table 5.8.

Table 5.8: Tuning the decay factor α for pre-training on Mini-ImageNet 5-way 1-shot classification. Accuracy is averaged across 100 randomly picked sub-networks.

α	0.1	0.5	5	10	50
Avg Accuracy	0.424	0.4145	0.5464	0.5323	0.4423

5.3.2.4 Layer-wise profiling

Hardware-aware NAS needs the run-time of sub-networks on the target hardware to guide the search process [18, 181]. However, the profiling stage can be time-consuming if given a low-end hardware as the profiling target and the search space is large. For instance, running a single network inference of VGG9 on the Raspberry Pi Zero with a 1GHz single-core ARMv6 CPU takes around 2.365 seconds to finish. If I assume this is the averaged time needed for profiling a sub-network, given that the entire search space includes around 10^9 sub-networks, a naive traverse will take a formidable amount of time which is approximately 6×10^5 hours. More importantly, the amount of profiling time scales with the number of hardware devices ($\mathcal{O}(H)$).

Existing hardware-aware NAS schemes build predictive methods to estimate the run-time of sub-networks [18, 181] and have a relatively significant error of around 10-20%. I show that, performing an exact profiling can be done with a low cost if allowing a per-layer profiling strategy.

Theoretically, if consider the original search space discussed in Section 5.3.2.2, profiling per-layer will in total run inference for $126 \times 4 = 504$ layers with various configurations. This is a much lower cost compared to brute-force profiling of all possible 10^9 sub-networks. Practically, on a Pi Zero, the consumed profiling time is 82 minutes, much lower than the 6×10^5 hours brute-force profiling time.

I re-implemented the latency predictor in OFA [18] to illustrate how a layer-wise

Table 5.9: Comparing latency predictor with our proposed profiling. MSE Error is the error between estimated and measured latency, Time is the total time taken to collect and build the estimator.

Hardware	Metric	Latency Predictor	Layer-wise Profiling
2080 Ti GPU	MSE Error	0.0188	0.00690
	Time	16.09 mins	6.216 secs
Intel i9 CPU	MSE Error	0.165	0.0119
	Time	21.92 mins	16.41 secs
Pi Zero	MSE Error	NA	0.00742
	Time	NA (Approx. 220 hours)	82.41 mins

profiling and look-up method is a perfect match in our learning scenario. I pick 16K training samples and 10K validation samples to train and test the latency predictor. This training setup is identical to Cai *et al.* [18]. I use another 10K testing samples to evaluate the performance of OFA-based latency predictor against the layer-wise profiling on different hardware systems in terms of MSE (measuring the latency estimation quality) and Time (measuring the efficiency).

As illustrated in Table 5.9, layer-wise profiling saves not only time but also has a smaller MSE error compared to a predictor-based strategy that is very popular in today's evolutionary-based NAS frameworks [18, 17]. In addition, layer-wise profiling shows orders of magnitude better run-time when targeting hardware devices with scarce computational resources. If I consider an IoT class device as a target (i.e the Raspberry Pi Zero), it requires an unreasonably large amount of time to generate training samples for latency predictors, making them an infeasible approach in real life. For instance, the total time consumed by latency predictor is infeasible to execute on Pi Zero (last row in Table 5.9). Of course, in reality, there is also a great number of IoT devices using more low-end CPUs compared to Pi Zero (ARMV5 or ARMV4), making the latency predictor even harder to be deployed on these devices. Also in a many-hardware setup considered in this paper, this profiling is executed O(H) times.

Most existing layer-wise look-up approaches consider at most mobile systems as targeting platforms [181, 182]. These systems are in general more capable than a great range of IoT devices. In this work, I demonstrate the effectiveness of this approach on more low-end systems (Raspberry Pi and Pi Zeros), illustrating this is the more scalable approach for hardware-aware NAS on constrained hardware systems.

5.3.2.5 Adaption strategy

The adaption strategy uses a genetic algorithm [171] to pick the best suited sub-network with respective to a given hardware constraint. In general, the adaption algorithm randomly samples a set of tasks from $\mathbb{T}_{val}$, and uses the averaged loss value and satisfaction to the hardware constraints as indicators the for the genetic algorithm. The genetic algorithm has a pool size P and number of iterations M, I demonstrate the optimal values are $P = 100, M = 200$ by sweeping different combinations of these hyper-parameters. I detailed this hyper-parameter tuning process in later paragraphs.

Algorithm 3 details the adaption algorithm. In the *Mutate* function, each architecture is ranked with the averaged loss across all sampled tasks, and 10% of the architectures with the lowest loss values are then used to perform a classic genetic algorithm mutation [171]. The mutation will allow the top-performing architectures to have two randomly picked architectural choices being modified to another choice that is not the original one. The mutation function considers the original pool of architectures ($\mathbb{A}$) and their averaged loss values (L_a). The cost of each architecture can be computed by the pre-build hardware-specific hash-table $H_t(\mathcal{A})$. I then only mutate the subset in $\mathbb{A}$ that their hardware cost has satisfied the constraints $\{\mathcal{A}|\mathcal{A} \in \mathbb{A} \wedge H_t(\mathcal{A}) \leq C\}$. The mutation is to randomly pick two options in the entire architectural space and change them to other choices that are different from the original.

I identify the following two hyperparameters that can potentially affect the performance, namely the number of iterations M and the pool size P, and then run an hyper-parameter analysis in Figure 5.8. The horizontal axis shows the number of iterations and the vertical axis shows the averaged accuracy on the sampled tasks for all architectures in the pool. Figure 5.8 shows that the accuracy convergence is reached after around 150 iterations, and running for additional iterations only provides marginal accuracy gains. For this reason, I picked the number of iterations to be 200 for a balance between accuracy and run-time. In the meantime, I notice in general a higher pool size will give better adapted accuracy. However, this does not mean the final searched accuracy is affected to the same degree. The final re-trained accuracy of searched architectures show an accuracy gap of 0.21% between $P = 100$ and $P = 200$ and 0.32% between $P = 100$ and $P = 500$. An increase in pool size can prolong the run-time significantly, I thus picked a pool size of 100 since it offers the best balance between accuracy and run-time.

5.3.2.6 NAS backbone design

One particular problem in few-shot learning is that models are prone to over-fitting. This is because only a small number of training samples are available for each task and the network normally iterate on these sample many times. I would like to explore on the architectural space to help models to overcome over-fitting and conduct a case study

Algorithm 3 The adaption algorithm

```
Input: $M$, $P$, $C$, $H_t$
$\mathbb{A} = \text{Init}(P)$                          ▷ Initialise a set of architectures with a size of $P$
for $i = 0$ to $M - 1$ do
    $L_a = \emptyset$
    $\mathbb{T}_s \sim p(\mathbb{T}_{val})$                     ▷ Obtain a subset from the validation task set
    for $\mathcal{A} \in \mathbb{A}$ do
        $L_t = \emptyset$
        for $\mathcal{T} \in \mathbb{T}_s$ do
            $l = \mathcal{L}(\mathcal{T}, \mathcal{A})$                                   ▷ Compute loss
            $L_t = L_t \bigcup \{l\}$
        end for
        $L_a = L_a \bigcup \{mean(L_t)\}$           ▷ Collect averaged loss values across all tasks
    end for
    $\mathbb{A} = Mutate(\mathbb{A}, L_a, H_t, C)$ ▷ Mutate the architectures based on hardware constraints
end for
```

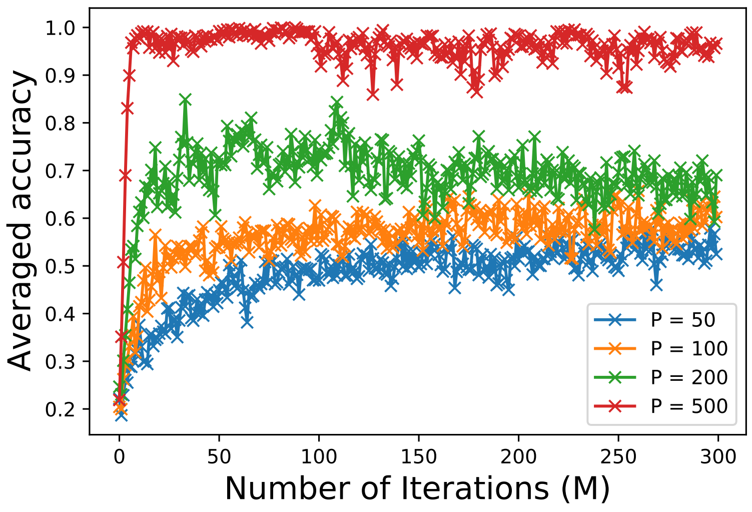


Figure 5.8: The effect of different pool size (P) and different number of iterations (M).

for different design options available for the backbone network. I identify the following key changes to the NAS backbone to help the models to have high accuracy in few-shot learning:

- $n \times n$ pooling: Pooling that applied to the final convolutional operation, $n \times n$ indicates the height and width of feature maps after pooling.
- Global Expansion (GE): Allowing the NAS to globally expand or shrink the number of channels of all layers.
- Adaptive Number of Layers (ANL): Allowing the NAS to use an arbitrary number of layers, the network then is able to early stop using only a fewer number of layers.

Figure 5.9 further illustrate that GE and ANL can allow a much smaller model compared to existing NAS backbones. The results in Table 5.10 suggest that these changes on a backbone will allow the search space to reach a much smaller model and then provide a better accuracy. In addition, Table 5.10 also illustrates that 5×5 pooling is necessary for a higher accuracy. I hypothesize this is because a relatively large fully-connected layer after the pooling is required for the network to achieve good accuracy in this meta-learning setup.

Table 5.10: A case study of different design options for the NAS backbone network. Experiments are executed with a model size constraint of $70K$ on the Mini-ImageNet 5-way 1-shot classification task.

Design options	Accuracy
MAML [47]	48.70%
MAML++ [8]	52.15%
1×1 Pool	42.28%
5×5 Pool	46.13%
5×5 Pool + GE	53.09%
5×5 Pool + GE + ANL	56.35%

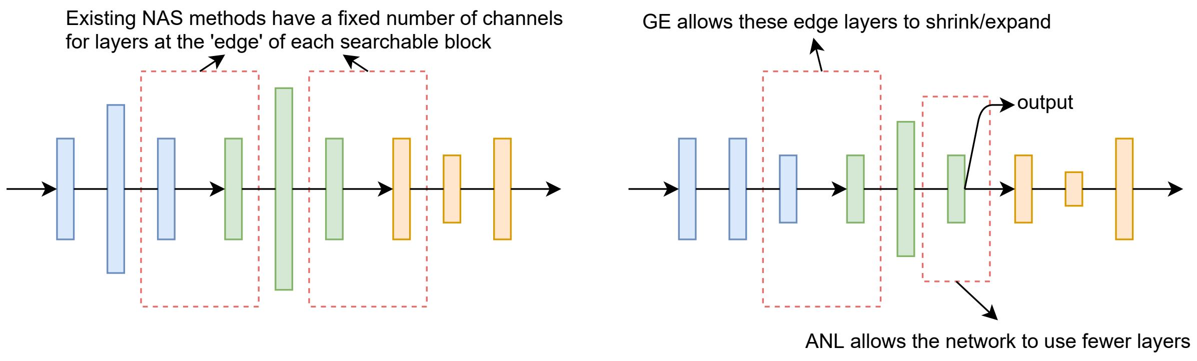

Existing NAS methods have a fixed number of channels for layers at the 'edge' of each searchable block
GE allows these edge layers to shrink/expand
output
ANL allows the network to use fewer layers

Figure 5.9: A graphical illustration of GE and ANL. Both methods will allow a more drastic change in model capabilities, allowing the searched model to deal with tasks with varying difficulties.

5.3.3 Evaluating H-Meta-NAS

I evaluate H-Meta-NAS in a few-shot learning setup. For each dataset, I search for the meta-architecture and meta-parameters. I then adapt the meta-architecture with respect to a target hardware-constraint pair. In the evaluation stage, I re-train the obtained hardware-aware task-specific architecture to convergence and report the final accuracy.

I consider three popular datasets in the few-shot learning community: Omniglot, Mini-ImageNet and Few-shot CIFAR100. I use the PytorchMeta framework to handle the datasets [34].

Omniglot is a handwritten digits recognition task, containing 1623 samples and 20 samples per class [96]. I use the meta train/validation/test splits used Vinyals *et al.* [166]. These splits are over 1028/172/423 classes (characters).

Mini-ImageNet is first introduced by Vinyals *et al.*. This dataset contains images of 100 different classes from the ILSVRC-12 dataset [37], the splits are taken from Ravi *et al.* [135].

FC100 is introduced by Oreshkin *et al.* [127], the datasets has 100 different classes from the CIFAR100 dataset [92]

Table 5.11 details the systems and representative devices considered. I use the ScaleSIM cycle-accurate simulator [139] for the Eyeriss [25] accelerator. Details about this simulation and more information with respect to the datasets and search configurations are in our Appendix.

Table 5.11: Details of hardware systems experimented with H-Meta-NAS.

System	Device	Specs
Cloud	Nvidia GeForce RTX 2080 Ti	4352 CUDA cores at 1635MHz
Mid-end CPU	Intel CPU	2.3GHz 8-core i9 CPU (2.3GHz, 16GB RAM)
Mobile CPU	Raspberry Pi 4B	Quad core Cortex-A72 (ARMv8) CPU (1.5GHz, 4GB RAM)
IoT	Raspberry Pi Zero	Single core ARM11 (ARMv6) CPU (1GHz, 512MB RAM)
ASIC	Eyeriss [25]	65nm with a core frequency of 250 MHz

5.3.3.1 Evaluating H-Meta-NAS searched architectures

Table 5.12 displays the results of H-Meta-NAS on the Omniglot 20-way 1-shot and 5-shot classification tasks. I match the size of H-Meta-NAS to MAML and MAML++ for a fair comparison. H-Meta-NAS outperforms all competing methods apart from the original MAML++. MAML++ uses a special evaluation strategy, it creates an ensemble of models with best validation-set performance. MAML++ then picks the best model from the ensemble based on support set loss and report accuracy on the query set. I then locally replicated MAML++ without this trick, and show that H-Meta-NAS outperforms it by a significant margin (+1.01% on 1-shot and +0.11% on 5-shot) with around half of the MACs ($4.95G$ compared to $10.07G$).

Table 5.13 shows the results of running the 5-way 1-shot and 5-shot Mini-ImageNet tasks, similar to the previous results, I match the size of searched networks to MAML and MAML++. Table 5.13 not only displays results on MAML methods with fixed-architectures, it also shows the performance of searched networks including Auto-Meta

Table 5.12: Results of Omniglot 20-way few-shot classification. We keep two decimal places for our experiments, and keep the decimal places as it was reported for other cited work. * reports a MAML replication implemented by Antoniou *et al.* [8]. Details are discussed in Section 5.3.3.1.

Method	Size	MACs	Accuracy 1-shot	Accuracy 5-shot
Siamese Nets [91]	$35.96M$	$1.36G$	99.2%	97.0%
Matching Nets [166]	$225.91K$	$20.29M$	93.8%	98.5%
Meta-SGD [106]	$419.86K$	$46.21M$	$95.93\% \pm 0.38\%$	$98.97\% \pm 0.19\%$
MAML [47]	$113.21K$	$10.07M$	$95.8\% \pm 0.3\%$	$98.9\% \pm 0.2\%$
MAML* (Replication from [8])	$113.21K$	$10.07M$	$91.27\% \pm 1.07\%$	98.78%
MAML++ * [8]	$113.21K$	$10.07M$	$\mathbf{97.65\% \pm 0.05\%}$	$\mathbf{99.33\% \pm 0.03\%}$
MAML++ (Local Replication)	$113.21K$	$10.07M$	$96.60\% \pm 0.28\%$	$99.00\% \pm 0.07\%$
H-Meta-NAS	**110.73K**	**4.95M**	$97.61 \pm 0.03\%$	$99.11\% \pm 0.09\%$

Table 5.13: Results of Mini-ImageNet 5-way classification. We use two decimal places for our experiments, and keep the decimal places of cited work as they were originally reported. T-NAS uses the complicated DARTS cell [107], it has a smaller size but a large MACs usage. Details are discussed in Section 5.3.3.1.

Method	Size	MACs	Accuracy 1-shot	Accuracy 5-shot
Matching Nets [166]	$228.23K$	$200.31M$	$43.44 \pm 0.77\%$	$55.31 \pm 0.73\%$
CompareNets [156]	$337.95K$	$318.38M$	$50.44 \pm 0.82\%$	$65.32 \pm 0.70\%$
MAML [47]	**70.09K**	$57.38M$	$48.70 \pm 1.84\%$	$63.11 \pm 0.92\%$
MAML++ [8]	**70.09K**	$57.38M$	$52.15 \pm 0.26\%$	$68.32 \pm 0.44\%$
Auto-Meta [88]	98.70K	-	$51.16 \pm 0.17\%$	$69.18 \pm 0.14\%$
BASE (Softmax) [144]	$1200K$	-	-	$65.4 \pm 0.7\%$
BASE (Gumbel) [144]	$1200K$	-	-	$66.2 \pm 0.7\%$
T-NAS * [107]	$24.3/26.5K$	$37.96/52.63M$	$52.84 \pm 1.41\%$	$67.88 \pm 0.92\%$
T-NAS++ * [107]	$24.3/26.5K$	$37.96/52.63M$	$54.11 \pm 1.35\%$	$69.59 \pm 0.85\%$
H-Meta-NAS	$70.28K$	**24.09M**	$\mathbf{57.36 \pm 1.11\%}$	$\mathbf{77.53 \pm 0.77\%}$

[88], BASE [144] and T-NAS [107]. H-Meta-NAS shows interesting results when compared to T-NAS and T-NAS++. H-Meta-NAS has a much higher accuracy (+3.26% in 1-shot and 7.94% in 5-shot) and a smaller MAC count, but uses a greater amount of parameters. T-NAS and T-NAS++ use DARTS cells [113]. This NAS cell contains a complex routing of computational blocks, making it not suitable for latency critical applications. I will demonstrate in Section 5.3.3.2 how this design choice gives a worse on-device latency performance.

I also show how H-Meta-NAS work with FC100. In Table 5.14, I further demonstrate the effectiveness of the proposed H-Meta-NAS on the FC100 dataset. T-NAS did not report their model sizes on this task, and our results suggest that H-Meta-NAS achieves the best accuracy on both the 1-shot and 5-shot setups.

Table 5.14: Results of FC100 5-way few-shot classification. I keep two decimal places for our experiments, and keep the decimal places of cited work as they were originally reported.

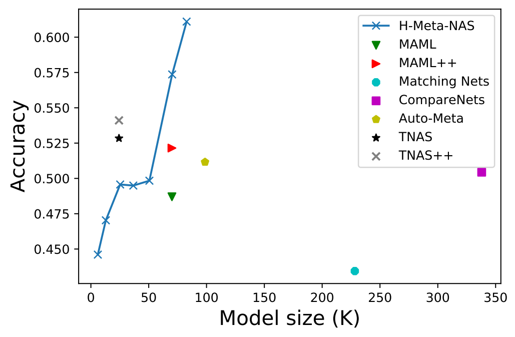
Method	Size	Accuracy 1-shot	Accuracy 5-shot
MAML	$70.09K$	$38.1 \pm 1.7\%$	$50.4 \pm 1.0\%$
MAML++	$70.09K$	$38.7 \pm 0.4\%$	$52.9 \pm 0.4\%$
T-NAS	-	$39.7 \pm 1.4\%$	$53.1 \pm 1.0\%$
T-NAS++	-	$40.4 \pm 1.2\%$	$54.6 \pm 0.9\%$
H-META-NAS	$55.52K$	$43.29 \pm 1.22\%$	$56.86 \pm 0.76\%$

5.3.3.2 H-Meta-NAS for diverse hardware platforms and constraints

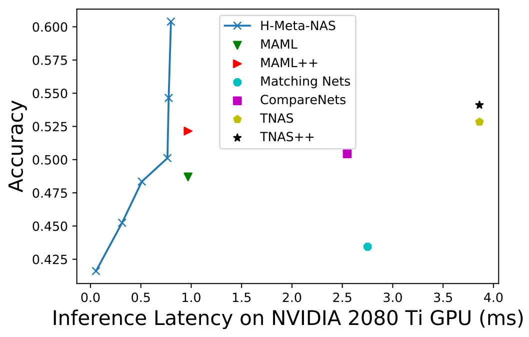
In addition to using the model sizes as a constraint for H-Meta-NAS, I use various latency targets on various hardware platforms as the optimisation target. Figure 5.10 shows how model sizes and GPU latencies can be used as constraints. T-NAS and T-NAS++ show a better performance on the size-accuracy plot in Figure 5.10a. However, the smaller model sizes of T-NAS do not provide a better run-time on GPU devices (Figure 5.10b), in fact, T-NAS based models have the worst run-time on GPU devices due to the complicated dependency of DARTS cells. Figure 5.11 illustrates the performance of H-Meta-NAS on different CPU devices and an ASIC hardware. The details of these hardware are described in Table 5.11. In Figure 5.11, H-Meta-NAS shows a better Pareto-frontier performance compared to a range of baselines and searched models. I only compare to MAML and MAML++ when running on Eyeriss due to the limitations of the ScaleSIM simulator [139]. Our results in both Figure 5.10 and Figure 5.11 demonstrate that H-Meta-NAS consistently generates more efficient models compared to various MAML-based methods.

I mostly follow the experiment setup in MAML++ [8]. In the pre-training stage, I train for 100 epochs, each epoch consists of 500 iterations. I also pick 600 tasks to be validation tasks. In the adaption stage, I randomly sample from the validation set, and pick 16 tasks to build a data slice for the architectures to traverse. In the final re-training stage of a searched architecture, we follow the strategy used in MAML++ [8]. I then introduce the detailed special configurations for the datasets:

- Omniglot: I randomly split 1200 characters for training, and the rest is used for testing. The images are augmented with randomised rotation of multiples of 90 degrees.

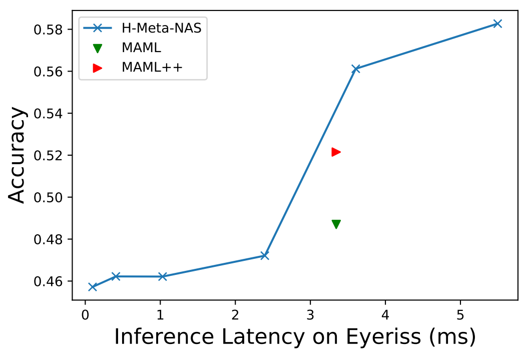


(a) Targeting model sizes

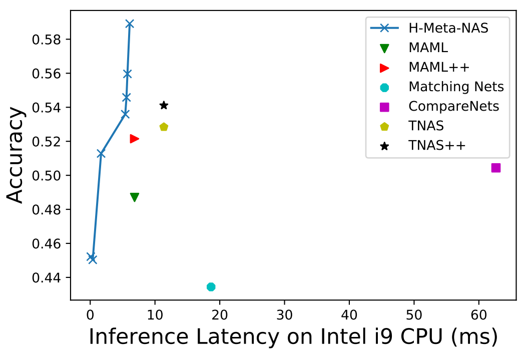


(b) Targeting a GPU

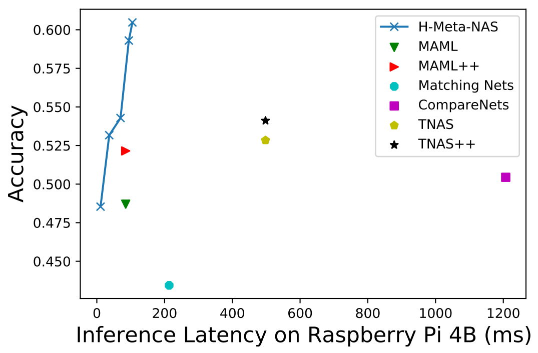
Figure 5.10: Applying H-Meta-NAS with model size and GPU latency as optimisation targets.



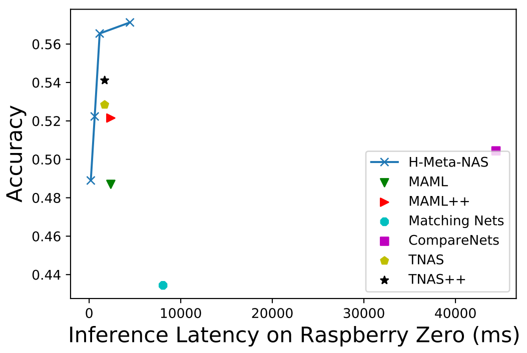
(a) Targeting an ASIC accelerator



(b) Targeting a mid-end CPU



(c) Targeting a low-end CPU



(d) Targeting an IoT device

Figure 5.11: Applying H-Meta-NAS with ASIC and CPUs as targets.

- Mini-ImageNet: All images are down-sampled to 84×84.

I use the ScaleSim framework [139] for simulating the Eyeriss [27] accelerator. ScaleSim is an open-source cycle-accurate CNN simulator. The simulator has certain limitations with respect to the DRAM simulation, it could be advanced with an external DRAM simulator but will cause a large run-time. So I kept the original setup and the DRAM simulation would report a read/write bandwidth requirements. For simplicity, I assume these DRAM requirements are met. In addition, it is a well-known fact that cycle-accurate simulators are slow to execute. Due to this reason, I only launched the MAML and MAML++ networks in the ScaleSim simulator.

5.3.3.3 A more complex NAS backbone

Table 5.15 shows how H-Meta-NAS performs with a more complicated NAS backbone. In previous experiments, I build the NAS on top of a VGG9 backbone since it is the architecture utilised in the MAML++ algorithm. For the purpose of having a fair comparison, I did not manually pick a complex NAS backbone. However, I demonstrate, in this section, that H-Meta-NAS can be applied with a more complicated backbone and it shows better final accuracy as expected. The trained accuracy of searched networks using ResNet12 reaches a 7.31% higher accuracy compared the original VGG9 backbone. In addition, I compare the proposed approach with state-of-the-art Metric-based meta-learning methods [191, 26]. Although using only a single inference pass (our method does not conduct inference runs on the support set when deployed), H-Meta-NAS shows competitive results with SOTA Metric-based methods while having a much smaller MACs usage (around 20×).

Table 5.15: Applying H-Meta-NAS to different NAS backbones and algorithms for the Mini-ImageNet 5-way 1-shot classification task.

Method	Network Backbone	Inference Style	Size	MACs	Accuracy
MAML [47]	VGG-based	Single Pass	$70.09K$	$57.38M$	$48.70 \pm 1.84\%$
MAML++ [8]	VGG-based	Single Pass	$70.09K$	$57.38M$	$52.15 \pm 0.26\%$
Meta-Baseline [26]	ResNet-based	Multi Pass	$12.44M$	$56.48G$	$63.17 \pm 0.23\%$
DeepEMD [191]	ResNet-based	Multi Pass	$12.44M$	$56.38G$	$65.91 \pm 0.82\%$
H-Meta-NAS	VGG-based	Single Pass	$70.28K$	$24.09M$	$57.36 \pm 1.11\%$
H-Meta-NAS	ResNet-based	Single Pass	$70.62K$	$28.19M$	$64.67 \pm 2.03\%$

5.3.3.4 Search complexity and search time

In Table 5.16, I show a comparison between H-Meta-NAS and various NAS schemes in the many-task many-device setup. Specifically, I consider a scenario with 500 tasks and 10

Table 5.16: Comparing the NAS search complexity with N tasks, H hardware platforms and C constraints. Search time is estimated for a deployment scenario with 500 tasks and 10 hardware-constraint pairs, estimation details are discussed in Appendix.

Method	Style	Hardware-aware	Search complexity	Search time (GPU hrs)
DARTS [113]	Gradient-based, single task	No	$\mathcal{O}(THC)$	$\approx 10^6$
Once-for-all [18]	Evolution-based, single task	Yes	$\mathcal{O}(N)$	$\approx 10^4$
TNAS & TNAS++ [107]	Gradient-based, multi task	No	$\mathcal{O}(HC)$	$\approx 10^3$
H-Meta-NAS	Evolution-based, multi task	Yes	$\mathcal{O}(1)$	40

different hardware-constraint paris. In this case, search complexity refers to the amount of time spent on searching for a new architecture. Our results in Table 5.16 suggest that H-Meta-NAS is the most efficient search method because of its low search complexity.

Due to the limited computing facilities available, I estimate the search time of DARTS [113], Once-for-all [18] and T-NAS [107] in a multi-task multi-device setup. I take the search time reported in the original publications and multiply them by the appropriate scaling factors. For DARTs, I take the search time (4 GPU days = 96 GPU hours) and multiply it by $H \times T = 5000$. I additionally assume a linear scaling relationship between search time and number of input image pixels, so I multiply the total search time by $\frac{84\times84}{32\times32}$, this gives us in total a search time of around 10^6. I perform the same estimation for Once-for-all [18] and T-NAS [107].

5.4 Summary

In this chapter, in Section 5.2, I propose a novel single-path, one-shot and gradient-based NAS algorithm named LPGNAS. LPGNAS is able to generate compact quantised networks with state-of-the-art accuracy. To my knowledge, this is the first piece of work that systematically studies the joint effect of quantisation and model architectures of GNNs in a NAS setup. I define the GNN quantisation search space and show how it can be co-optimised with the original architectural search space. The end results demonstrate that a co-optimisation between the architectural and quantisation spaces greatly improves network accuracy. The searched networks show Pareto dominance on a accuracy model size trade-off over all manually designed networks. In addition, I empirically show the limitation of GNN quantisation using the proposed NAS algorithm, most of the searched networks converge to a 4-bit weight 8-bit activation quantisation setup. Despite this limitation, LPGNAS demonstrates superior performance when compared to networks generated manually or by other NAS methods on a wide range of datasets.

In Section 5.3 of this chapter, I show H-Meta-NAS, a NAS method focusing on fast adaption of not only model weights but also model architectures in a many-task many-device few-shot learning setup. H-Meta-NAS shows a Pareto dominance when compared to a wide range of MAML baselines and other NAS results. I study the effectiveness of H-Meta-NAS on a wide variety of hardware systems and constraints, and demonstrate its superior performance on real-hardware devices using an orders of magnitude shorter search time compared to existing NAS methods.

Chapter 6

Conclusion

The deep learning approach offers state-of-the-art performance in many applications and is now widely deployed in many of today's hardware architectures. As illustrated in this dissertation, the optimisation space of DNN inference covers both the software and hardware stacks. In this dissertation, I provided a set of compression techniques on the software stack for lowering the computational and memory requirements of running DNN inference (Chapter 3). I also demonstrate specialised hardware can successfully accelerate DNNs in Chapter 4. At the end, I demonstrate how automated machine learning, or in particular network architecture search, can be made hardware-aware for finding efficient models for emerging network types and in new learning setups (Chapter 5).

I would like address these interesting observations:

- *Software optimisation, such as compression methods, will affect the efficiency of running ML workloads on hardware, but this design space is determined by the capabilities of the given hardware. In the meantime, these compression methods also reshape today's hardwares.*

  The chosen hardware platform decides the applicability of the software optimisations. For instance, GPUs struggle to realise the theoretical performance gains of fine-grained pruning due to limitations of the SIMT architecture [49], but bit-serial DNN accelerators can utilise not only element-wise but also bit-wise sparsities. In practice, users have to be aware of the underlying hardware system when selecting software compression methods, this suggests that DNN implementation frameworks should carefully select optimisation techniques based on the target hardware. In this dissertation, I demonstrated that dynamic channel pruning is particularly useful on CPU devices, but cannot reach its theoretical gains on GPUs because of the low efficiency of scatter and gather operations of GPUs. In addition, the focused quantisation scheme presented is only applicable on custom hardware. It is also interesting to notice that the software-level optimisations for ML workloads are also reshaping today's hardware architectures. For instance, NVIDIA GPUs now have tensor cores [118] and Intel CPUs [2] have extended instructions to support low-precision matrix multiplications. Architecture and micro-architectures are adapting their target markets (or workloads) and are evolving all the time in order to catch the performance scaling.

- *Algorithmic changes offer larger performance gains than pure hardware modifications, co-design offers multiplying gains. However, all of these modifications might have diminishing returns after a while if we try to specilise further.*

  From the experiments conducted in this dissertation, I observe there are huge performance gains possible from purely software optimisation, in general, these gains outweigh those possible from hardware optimisations alone, but co-design often provides the best performance.

  This is an interesting phenomenon for hardware designers, the overall design for DNN accelerators should be software orientated and co-design focused.

  In addition, the possible returns from both hardware and software optimisations might have a limit. Previous research in the filed of network compression shows great compression rates but many of these proposed methods optimise on overlapped design spaces. In other words, these methods are not orthogonal to each other. For instance, the proposed dynamic channel pruning shows an around 3× compression ratio for VGG16 but only 1.98× for ResNet18 with no accuracy loss, this implies that pruning is not totally orthogonal to architectural engineering. The latter apparently has a more carefully designed, compact architecture so that it benefits less from network pruning.

- *Reconfigurable hardware enlarges the design space and provides significant performance gains.*

  In Chapter 4, I utilised a flexible hardware design consisting of many small cores. This design in turn provides a larger quantisation optimisation space. I think this offers an interesting trade-off from the co-design point of view: complex hardware might provide a large design space but also enables a more involved exploration on the software stack.

- *Is there a general AI accelerator? New network types and structures can be difficult for DNN accelerators, specialised for a narrower range of models, to process efficiently.*

  There was a long-held belief in the community that we can build a 'general AI accelerator' and running AI applications is nothing more than accelerating dense matrix multiplications. I would like to argue that this belief might be true when we are in the CNN era, but now it is less likely to be the case when we are facing complex models such as Transformers and GNNs. The diverse structures of DNNs make building a universal accelerator almost impossible. Specialised accelerators for CNNs can not exploit sparsities that exist in GNNs or NLP models. This indicates that the reconfigurability of hardware is an interesting alternative to accelerating a diverse set of models. FPGA-like reconfigurability comes at a very high cost. New devices or technologies, such as memristor-based accelerators [187], might offer the potential for very low-cost reconfigurability.

- *NAS algorithms benefit from search strategies that are specialised for particular use-cases and applications.*

  Although Network Architecture Search algorithms show promising performance in many domains, these algorithms normally require special treatments to fit to different application scenarios. Chapter 5 shows two NAS algorithms, namely the LPGNAS and H-Meta-NAS. The search of the routing strategy is crucial for LPGNAS, and the proposed Global Expansion and Adaptive Number of Layers is the key for H-Meta-NAS to achieve high accuracy on few-shot learning tasks. These application specific modifications are necessary for the NAS techniques to achieve high accuracy.

6.1 Future research

6.1.1 Software-hardware co-design network architecture search

In Chapter 5, I presented a flexible hardware accelerator that can adapt to different layer-wise quantisation strategies, this does not fully capture the large software-hardware co-design space of DNN inference acceleration. This dissertation considers a search space of quantisation choices of a streaming-based architecture on FPGAs, however, software and hardware co-design architecture search methods have been established on systolic-array based accelerators [4, 211]. These co-design search algorithms normally operate with a limited number of architectural hyper-parameters on systolic-array based architectures. The hardware search space of these algorithms does not allow drastic hardware transformations such as transforming from a streaming-based accelerator to a systolic array based accelerator. The exploration of the hardware search space is still limited in current co-design NAS methods.

One of the root causes of this limitation is the difficulties of applying transformations in the hardware space. Unlike software engineering, code can be transformed easily in intermediate representations, there lacks a concise yet powerful hardware representation that supports flexible transformations. The existing MLIR [99] is built for compiler optimisations from a software engineering perspective. Ideally, hardware research in this field would like to see a prototype hardware design that can be applied with various transformations to sweep a much larger hardware design space than only the number of PEs or number of multipliers. These hardware transformations can then be jointly searched with architectural choices to offer a much larger exploration in the co-design search space.

Another interesting challenge when performing HW/SW co-design optimisations stems from the need to perform retraining. The optimisation covers a range of lossy techniques, these methods reduce the run-time costs but might induce accuracy degradation. This phenomenon changes the optimisation problem to be drastically different from classic software compilation flow. In a standard software compiler, such as LLVM [98], there are various methods to guarantee the final execution result is unchanged after a series of complex code transformations. I realised it is hard to fulfill this guarantee when optimising DNN inference, unless re-training is allowed in the optimisation flow. However, re-training DNNs then introduces additional problems such as the prolonged optimisation time.

Another fundamental challenge in this co-design is the lack of abilities to measure hardware performance in a quick and accurate manner. We have seen LLVM using local profiling information to help the tool to further optimise code snippets [98]. Ideally, if we can quickly profile hardware platforms, there is a chance to build hardware-aware optimisations in this co-design circle. However, this performance estimation is highly data-dependent if we consider different co-design targets. For instance, the optimisations applicable for different sets of neural networks can be vastly different. In addition, it is well-known that precise hardware simulation can be extremely slow, while coarse-grained power emulation can be very inaccurate.

6.1.2 General-purpose AI accelerator

I have already argued that the design of a universally optimal AI accelerator architecture is not feasible. Nevertheless, I believe there are a number of interesting directions that might help address the challenge.

- Multiple heterogeneous ML accelerators on one chip:

  Having a heterogeneous system might be an obvious approach, however, having an increasing number of ML accelerators on one chip will increase the design and integration complexities.

- Try to find a sweet-spot in terms of a 'general-purpose' ML accelerator:

  One of the most famous ML accelerators today is maybe Google's TPU chip. In its latest retrospective report [85], one of the interesting discussion is how to keep backwards compatibility. This then causes the newest TPU-V4 to not only support BERT and RNNs but also needs to keep serving MLP and CNNs from 2017. Google finds the sweet-spot of designing the ML accelerator by sweeping across Google's inference workloads, the most dominant neural network types then drive the entire architecture design.

- Extend general-purpose multi-core or many-core processors:

  Since ML workloads are getting more complex, one possible approach is to extend the instruction sets of CPUs to support ML applications [9]. The core matrix-multiply operations can be accelerated using special instructions while the exotic operations are well supported by general-purpose computation platforms.

All of these above-mentioned approaches are possible methods for building a general-purpose AI accelerator. The differences in energy profiles, costs, design complexities, *etc.* will then all affect the cost effectiveness of these solutions.

Recent trends in AI accelerators, however, are going in the opposite direction of building general-purpose AI cores. First, as illustrated in Figure 2.9, cores designed for applications (*e.g.* embedded systems or cloud systems) have a big performance gap. Since the power budget is different for various applications, micro-architectures of these cores are now vastly different. Second, DNN inference and training have totally different characteristics and the industry start to have individual designs for these two workloads. For instance, TPU-V4 surprisingly only supports DNN inference but V3 was able to run both training and inference [85].

It seems like the industry is choosing the path of building different accelerators for different DNN applications and they might start to bundle them on a system level depending on the task demands. These cores can be whether integrated in an SoC if it is for autonomous driving, or connected using fast communication links (fast rack-level switches or InfiniBand) in data centers.

6.1.3 Large AI chips

Starting from GraphCore AI, there is a rise of using larger die sizes for AI chips [86]. Most startup companies in this domain are using this non-traditional approach, such as Groq [60] and Cerebras [1]. The new release of Cerebras' second generation Wafer Scale Engine surprised the hardware design community by integrating a few thousands of compute cores on a massive 46000 mm^2 silicon, and popular NVIDIA GPU cores have a die size of around 500 - 700 mm^2. These large cores use the data-streaming architecture, where data is streamed to the chip and weights of the DNNs stay on-chip during the computation. This computation pattern is in fact very similar to Tomato (Chapter 4). Additionally, these accelerators have a large SRAM size at GB level, *e.g.* Cerebras contains 40GB SRAM.

The aggressively large chip sizes and dataflow hardware design then offer a great challenge to software designers. Many designers are thinking of supporting training on these platforms, the optimisation targets and possible transformations are different from traditional GPU programming. For instance, supporting dynamic operators in a computational graph might be challenging for these streaming based hardware accelerators, do programmers now have more restrictions in their high level languages or compiler engineers can invent smart ways of converting these dynamic operations to static ones? It is interesting to see how future research in ML compilers can equip these hardware accelerators with more capabilities.

6.1.4 Handling complex data types in Machine Learning using custom hardware

Most of todays' co-design frameworks in ML focus solely on one type of data. An interesting fact is that most data information in today's world contains more complex formats and relationships, *e.g.* multi-modal data connected in complex graphs. These sophisticated data structures can challenge the performance of existing learning systems. A common practice to obtain information through complex data structures is through feature engineering, so that these feature information fits into the ML pipeline that we are familiar with. Practitioners have to manually extract features from this unstructured combinatorial information, and create embeddings for learning methods. For example, most GNNs assume the graph nodes have embedding information trained from large scale feature extraction [189]. Theoretically speaking, building compact feature information relies on the manifold hypothesis. The hypothesis states that, empirically, high dimensional sampled data points will fit to low dimensional non-linear manifolds, generalizing the projection theorem in Hilbert space [44]. However, this feature learning process remains to be a lossy operation and might affect the data quality. One of the major reasons of keeping the unified compact data format is the lack of support of these complex data types in today's hardware systems. It is interesting to see how complex data types can be well supported on custom hardware, and how heterogeneous data information can be consumed by ML algorithms.

Appendix A

Structure of MCifarNet used in dynamic channel pruning

Figure A.1 shows how the skipping probabilites heat maps of the convolutional layer Conv4 evolve as we train MCifarNet. The network was trained for 12 epochs, and I saved the model at every epoch. The heat maps are generated with the saved models in sequence, where I apply the same reordering to all heat map channels with the sorted result from the first epoch. It can be observed that as we train the network, the channel skipping probabilites become more pronounced.

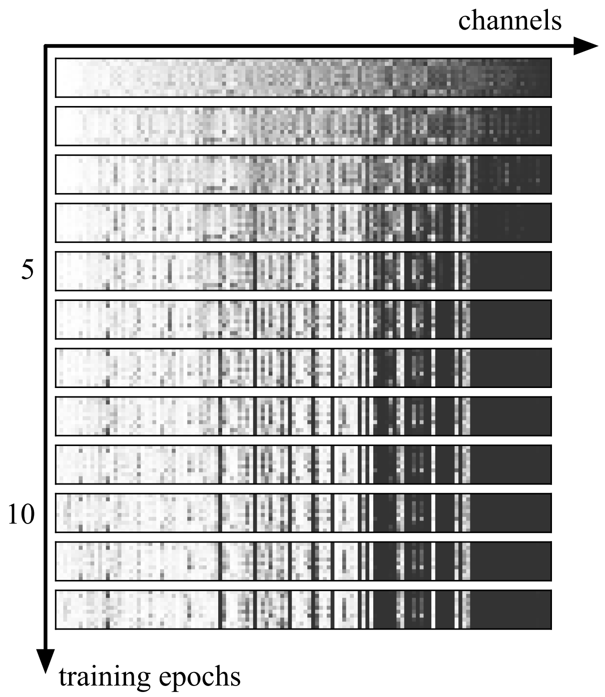


Figure A.1: The training history of a convolutional layer Conv4 in MCifarNet. The history is visualized by the 12 skipping probabilites heat maps, where the heights denote the 10 categories in Cifar10, and channels in Conv4 occupy the width.

Appendix B

Graph dataset details, graph sampler configurations and the grid search setup for baselines

In this section we decribe in details the dataset we used. In our experiments we use dataset preprocessing and loading implementations from Pytorch Geometric [45].

Citation Dataset: Citation dataset [184] is a standard benchmark dataset for graph learning. It comprises three publication datasets, which are Cora, CiteSeer and PubMed. There is an additional extended version of Cora called Cora-Full [13]. Cora contains all Machine Learning papers in the Cora-Full graph. In these datasets, nodes correspond to bag-of-word features of the documents and edges indicates citation. Each node has a class label. In this paper we use the 6:2:2 train/validation/test split ratio.

Amazon Dataset: Amazon datasets [146], including Amazon Photos and Amazon Computers, are segments of the Amazon co-purchase graph. Nodes represent goods while edges represent frequent co-purchase of linked goods. Node feature is bag-of-word features of product review, while node label is its category. We use the 6:2:2 train/validation/test split ratio.

Flickr and Yelp Dataset: Flickr and Yelp datasets are introduced along GraphSAINT [189]. In Flickr dataset, nodes represent images uploaded to Flickr, and edges represent sharing of common properties (e.g. location and comments by same user). Node feature is 500-dimensional bag-of-words representation based on SIFT descpritions, and node label is its class. In Yelp dataset, nodes represent users and edges represent friendship between users. Node feature is summed word2vec embeddings of words in the user's reviews, and node label is a multi-hot vector representing which types of businesses has the user reviewed. For both dataset we use the 6:2:2 train/validation/test split ratio.

For producing quantised baseline networks, we manually grid searched options listed in Table 5.3 and follow the order from bottom to top. We stop the search and retrieve to the previous quantisation stage if the current stage shows an accuracy drop of more than 0.5%. It is worth to mention this grid search for quantisation is very time-consuming, and we therefore only performed on Cora and used the found quantisation strategy for the rest of the Citation datasets.

Table B.1 shows the configurations of the baseline networks we've used in this paper.

Table B.1: Baseline networks configurations.

Networks	Layers	Channels
GAT	2	32
GAT-V2	2	64
JKNet	2	32
JKNet-V2	2	512
SageNet	2	16
SageNet-V2	2	512

Appendix C

Details of VGG9 and ResNet12 backbones used for H-Meta-NAS

The two tables below show the NAS backbones of H-Meta-NAS. Clearly the ResNet-based NAS backbone is significantly more complicated. The kernel size search space is $\{1, 3, 5\}$. The channel expansion search space is $\{0.25, 0.5, 0.75, 1, 1.5, 2, 2.25\}$ for the VGG-based NAS backbone but $\{0.25, 0.5, 1, 1.5, 1.75, 2\}$ for the ResNet-based backbone. The reason for the modification in search space is because the GPU RAM limitation does not support an expansion size of 2.25 on the ResNet-based backbone. The activation search space contains $\{['relu','elu','selu','sigmoid','relu6','leakyrelu'\}$.

Table C.1: Details of the VGG9 NAS backbone

Layer Name	Base channel counts	Stride
Layer0	64	2
Layer1	64	2
Layer2	64	2
Layer3	64	2

Table C.2: Details of the ResNet12 NAS backbone

Layer Name	Base channel counts	Stride
Block0_Layer0	32	2
Block0_Layer1	32	1
Block0_Layer2	32	1
Block1_Layer0	64	2
Block1_Layer1	64	1
Block1_Layer2	64	1
Block2_Layer0	128	2
Block2_Layer1	128	1
Block2_Layer2	128	1
Block3_Layer0	256	2
Block3_Layer1	256	1
Block3_Layer2	256	1

Bibliography

[1] The wafer-scale engine. `https://cerebras.net/chip/`. Accessed: 2021-08-20.

[2] Advanced matrix extension (AMX) - x86 - wikichip. `https://en.wikichip.org/wiki/x86/amx`. Accessed: 2021-09-21.

[3] Intel movidius myriad x vision processing unit. `https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu/movidius-myriad-x.html`. Accessed: 2021-08-20.

[4] Mohamed S Abdelfattah, Lukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D Lane. Codesign-nas: Automatic FPGA/CNN codesign using neural architecture search. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 315–315, 2020.

[5] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, et al. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE transactions on neural networks and learning systems*, 30(3):644–656, 2018.

[6] Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 2270–2278. 2016.

[7] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An architecture for ultralow power binary-weight CNN acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):48–60, 2018.

[8] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *International Conference on Learning Representations (ICLR)*, 2019.

[9] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman,

et al. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro*, 39 (2):29–36, 2019.

[10] Sercan O Arık and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. In *AAAI*, volume 35, pages 6679–6687, 2021.

[11] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[12] Lin Bai, Yiming Zhao, and Xinming Huang. A CNN accelerator on FPGA using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2018.

[13] Aleksandar Bojchevski and Stephan Günnemann. Deep gaussian embedding of graphs: Unsupervised inductive learning via ranking. *International Conference on Learning Representations (ICLR)*, 2018.

[14] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *International Conference on Learning Representations (ICLR)*, 2018.

[15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[16] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *International Conference on Machine Learning*, pages 678–687. PMLR, 2018.

[17] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *International Conference on Learning Representations (ICLR)*, 2019.

[18] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *International Conference on Learning Representations (ICLR)*, 2020.

[19] Pierre-Luc Carrier, Aaron Courville, Ian J Goodfellow, Medhi Mirza, and Yoshua Bengio. FER-2013 face database. *Technical report*, 2013.

[20] Francesco Paolo Casale, Jonathan Gordon, and Nicolo Fusi. Probabilistic neural architecture search. *arXiv preprint arXiv:1902.05116*, 2019.

[21] Stephen Cass. Taking ai to the edge: Google's tpu now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17, 2019.

[22] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *International Conference on Machine Learning*, 2018.

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: End-to-end optimization stack for deep learning. *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, 2018.

[24] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284, 2014.

[25] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017. ISSN 0018-9200. doi: 10.1109/JSSC.2016.2616357.

[26] Yinbo Chen, Zhuang Liu, Huijuan Xu, Trevor Darrell, and Xiaolong Wang. Meta-baseline: Exploring simple meta-learning for few-shot learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9062–9071, 2021.

[27] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[28] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Universal deep neural network compression. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):715–726, 2020.

[29] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Accelerating persistent neural networks at datacenter scale. In *Hot Chips*, volume 29, 2017.

[30] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *International Conference on Learning Representations (ICLR)*, 2015.

[31] Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. Xnor neural engine: A hardware accelerator ip for 21.6-fj/op binary neural network inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11): 2940–2951, 2018.

[32] Matthieu Courbariaux, Yoshua Bengio, et al. Training deep neural networks with low precision multiplications. In *International Conference on Learning Representations*, 2015.

[33] Jiequan Cui, Pengguang Chen, Ruiyu Li, Shu Liu, Xiaoyong Shen, and Jiaya Jia. Fast and practical neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6509–6518, 2019.

[34] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, and Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch, 2019. URL `https://arxiv.org/abs/1909.06576`. Available at: https://github.com/tristandeleu/pytorch-meta.

[35] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 749–763, 2019.

[36] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 1977.

[37] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, 2018.

[39] Xuanyi Dong, Junshi Huang, Yi Yang, and Shuicheng Yan. More is less: A more complicated network with less inference complexity. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[40] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold,

Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations (ICLR)*, 2021.

[41] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[42] Abhimanyu Dubey, Moitreya Chatterjee, and Narendra Ahuja. Coreset-based neural network compression. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 454–470, 2018.

[43] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[44] Charles Fefferman, Sanjoy Mitter, and Hariharan Narayanan. Testing the manifold hypothesis. *Journal of the American Mathematical Society*, 29(4):983–1049, 2016.

[45] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[46] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. PerforatedCNNs: Acceleration through elimination of redundant convolutions. In *Advances in Neural Information Processing Systems (NIPS)*, pages 947–955, 2016.

[47] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

[48] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale DNN processor for real-time ai. In *45th Annual International Symposium on Computer Architecture*, 2018.

[49] Wilson WL Fung and Tor M Aamodt. Thread block compaction for efficient simt control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 25–36. IEEE, 2011.

[50] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng-zhong Xu. Dynamic channel pruning: Feature boosting and suppression. *International Conference on Learning Representations (ICLR)*, 2019.

[51] Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. GraphNAS: Graph neural architecture search with reinforcement learning. *arXiv preprint arXiv:1904.09981*, 2019.

[52] Spyros Gidaris and Nikos Komodakis. Dynamic few-shot visual learning without forgetting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4367–4375, 2018.

[53] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

[54] Jianzhu Guo, Xiangyu Zhu, Chenxu Zhao, Dong Cao, Zhen Lei, and Stan Z. Li. Learning meta face recognition in unseen domains. *CoRR*, abs/2003.07733, 2020. URL `https://arxiv.org/abs/2003.07733`.

[55] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-Eye: A complete design flow for mapping CNN onto customized hardware. In *IEEE Computer Society Annual Symposium on VLSI*, 2016.

[56] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[57] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient DNNs. In *Advances in Neural Information Processing Systems*, 2016.

[58] Yong Guo, Yin Zheng, Mingkui Tan, Qi Chen, Jian Chen, Peilin Zhao, and Junzhou Huang. Nat: Neural architecture transformer for accurate and compact architectures. *Advances in Neural Information Processing Systems*, 32, 2019.

[59] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *European Conference on Computer Vision*, 2020.

[60] Linley Gwennap. Groq rocks neural networks. *Microprocessor Report, Tech. Rep., jan*, 2020.

[61] Philipp Gysel et al. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2018.

[62] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[63] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[64] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.

[65] Babak Hassibi, David G. Stork, and Gregory Wolff. Optimal brain surgeon: Extensions and performance comparisons. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems (NIPS)*, pages 263–270. 1994.

[66] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[68] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[69] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2234–2240, 2018.

[70] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, 2017.

[71] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[72] Geoffrey Hinton, Li Deng, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[73] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[74] Jui-Yang Hsu, Yuan-Jui Chen, and Hung-yi Lee. Meta learning for end-to-end low-resource speech recognition. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7844–7848. IEEE, 2020.

[75] Weizhe Hua, Christopher De Sa, Zhiru Zhang, and G. Edward Suh. Channel gating neural networks. *Advances in Neural Information Processing Systems*, pages 1886–1896, 2019.

[76] Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G Edward Suh. Boosting the performance of CNN accelerators with dynamic fine-grained channel gating. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 139–150, 2019.

[77] Weizhe Hua, Yuan Zhou, Christopher M De Sa, Zhiru Zhang, and G Edward Suh. Channel gating neural networks. In *Advances in Neural Information Processing Systems*, pages 1886–1896, 2019.

[78] Qiangui Huang, Kevin Zhou, Suya You, and Ulrich Neumann. Learning to prune filters in convolutional neural networks. In *IEEE Winter Conference on Computer Vision*. 2018.

[79] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[80] Itay Hubara, Matthieu Courbariaux, et al. Binarized neural networks. In *Advances in Neural Information Processing Systems*. 2016.

[81] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[82] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and −1. In *Signal Processing Systems*, 2014.

[83] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[84] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

[85] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i.

[86] Ilyes Kacher, Maxime Portaz, Hicham Randrianarivo, and Sylvain Peyronnet. Graphcore c2 card performance for image-based deep learning application: A report. *arXiv preprint arXiv:2002.11670*, 2020.

[87] Kari Kalliojarvi and Jaakko Astola. Roundoff errors in block-floating-point systems. *IEEE transactions on signal processing*, 44(4):783–790, 1996.

[88] Jaehong Kim, Sangyeul Lee, Sungwan Kim, Moonsu Cha, Jung Kwon Lee, Youngduck Choi, Yongseok Choi, Dong-Yeon Cho, and Jiwon Kim. Auto-meta: Automated gradient based meta learner search. *arXiv preprint arXiv:1806.06927*, 2018.

[89] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2016.

[90] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *Advances in neural information processing systems*, 30, 2017.

[91] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015.

[92] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[93] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 2012.

[94] Alex Krizhevsky et al. The CIFAR-10 dataset. *http://www.cs.toronto.edu/kriz/cifar.html*, 2014.

[95] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26 (2):203–215, 2007.

[96] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[97] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *International Conference on Learning Representations (ICLR)*, 2020.

[98] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[99] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[100] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems (NIPS)*, pages 598–605. 1990.

[101] Edward H. Lee, Daisuke Miyashita, et al. Lognet: Energy-efficient neural networks using logarithmic computation. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2017.

[102] Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with ADMM. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[103] Christian Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory*, pages 398–416. Springer, 1993.

[104] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[105] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. 2017.

[106] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-sgd: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*, 2017.

[107] Dongze Lian, Yin Zheng, Yintao Xu, Yanxiong Lu, Leyu Lin, Peilin Zhao, Junzhou Huang, and Shenghua Gao. Towards fast adaptation of neural architectures with meta learning. In *International Conference on Learning Representations*, 2019.

[108] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE Computer Society, 2019.

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

[110] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in neural information processing systems*, pages 345–353, 2017.

[111] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[112] Baoyuan Liu, Min Wang, et al. Sparse convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[113] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *International Conference on Learning Representations (ICLR)*, 2019.

[114] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *International Conference on Computer Vision (ICCV)*, 2017.

[115] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. ThiNet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5058–5066, 2017.

[116] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. ThiNet: A filter level pruning method for deep neural network compression. In *ICCV*, pages 5058–5066, 2017.

[117] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto FPGA. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–8. IEEE, 2016.

[118] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[119] Moritz B Milde, Daniel Neil, et al. ADaPTION: Toolbox and benchmark for training convolutional neural networks with reduced numerical precision weights and activation. *CoRR*, abs/1711.04713, 2017.

[120] Sparsh Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 97:428–442, 2019.

[121] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[122] Duncan Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H. W. Leong. A customizable matrix multiplication framework for the Intel HARPv2 Xeon + FPGA platform. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018.

[123] Tsendsuren Munkhdalai and Hong Yu. Meta networks. In *International Conference on Machine Learning*, pages 2554–2563. PMLR, 2017.

[124] Giridhar Sreenivasa Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran, and Ponnuswamy Sadayappan. Optimal loop unrolling for gpgpu programs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.

[125] Milos Nikolic, Mostafa Mahmoud, Andreas Moshovos, Yiren Zhao, and Robert Mullins. Characterizing sources of ineffectual computations in deep learning networks. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[126] Miloš Nikolić, Mostafa Mahmoud, Andreas Moshovos, Yiren Zhao, and Robert Mullins. Characterizing sources of ineffectual computations in deep learning networks. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 165–176. IEEE, 2019.

[127] Boris N Oreshkin, Pau Rodriguez, and Alexandre Lacoste. Tadam: Task dependent adaptive metric for improved few-shot learning. *Advances in neural information processing systems*, 31, 2018.

[128] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 27–40. ACM, 2017.

[129] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[130] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR, 2018.

[131] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *International Conference on Learning Representations*, 2018.

[132] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, and Sen Song. Going deeper with embedded FPGA platform for convolutional neural network. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[133] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9, 2019.

[134] Atul Rahman, Jongeun Lee, and Kiyoung Choi. Efficient FPGA acceleration of convolutional neural networks using logical-3d compute array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1393–1398. IEEE, 2016.

[135] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *International Conference on Learning Representations (ICLR)*, 2017.

[136] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[137] Lakshminarayanan Renganarayanan, Daegon Kim, Michelle Mills Strout, and Sanjay Rajopadhye. Parameterized loop tiling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):1–41, 2012.

[138] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017 (19):70–76, 2017.

[139] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic CNN accelerator simulator. *arXiv preprint arXiv:1811.02883*, 2018.

[140] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6): 52–59, 1997.

[141] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[142] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, 2019.

[143] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[144] Albert Shaw, Wei Wei, Weiyang Liu, Le Song, and Bo Dai. Meta architecture search. *arXiv preprint arXiv:1812.09584*, 2018.

[145] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *International Conference on Learning Representations (ICLR)*, 2017.

[146] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[147] Junzhong Shen, You Huang, Zelong Wang, Yuran Qiao, Mei Wen, and Chunyuan Zhang. Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018.

[148] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN accelerator efficiency through resource partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[149] Lei Shi, Yifan Zhang, Jian Cheng, and Hanqing Lu. Two-stream adaptive graph convolutional networks for skeleton-based action recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12026–12035, 2019.

[150] Sungho Shin, Kyuyeon Hwang, and Wonyong Sung. Fixed-point performance analysis of recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 976–980. IEEE, 2016.

[151] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[152] David So, Quoc Le, and Chen Liang. The evolved transformer. In *International Conference on Machine Learning*, pages 5877–5886. PMLR, 2019.

[153] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[154] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702, 2020.

[155] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[156] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1199–1208, 2018.

[157] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[158] Frederick Tung, Greg Mori, and Simon Fraser. CLIP-Q: Deep network compression learning by in-parallel pruning-quantization. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[159] Yash Ukidave, David Kaeli, Umesh Gupta, and Kurt Keville. Performance of the nvidia jetson tk1 in hpc. In *2015 IEEE International Conference on Cluster Computing*, pages 533–534. IEEE, 2015.

[160] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[161] Han Vanholder. Efficient inference with tensorrt, 2016.

[162] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30:5998–6008, 2017.

[163] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJXMpikCZ`.

[164] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2016.

[165] Thomas Verelst and Tinne Tuytelaars. Dynamic convolutions: Exploiting spatial sparsity for faster inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2320–2329, 2020.

[166] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. *Advances in Neural Information Processing Systems*, 2016.

[167] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The World Wide Web Conference*, pages 2022–2032, 2019.

[168] Yunchao Wei, Huaxin Xiao, Honghui Shi, Zequn Jie, Jiashi Feng, and Thomas S Huang. Revisiting dilated convolution: A simple approach for weakly-and semi-supervised semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7268–7277, 2018.

[169] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2074–2082. 2016.

[170] Wei Wen, Cong Xu, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Coordinating filters for faster deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 658–666, 2017.

[171] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[172] JH Wilkinson. Rounding errors in algebraic processes. 1965.

[173] Felix Winterstein, Samuel Bayliss, and George A Constantinides. High-level synthesis of dynamic data structures: A case study using vivado hls. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 362–365. IEEE, 2013.

[174] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

[175] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[176] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*, 2017.

[177] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *International Conference on Learning Representations (ICLR)*, 2019.

[178] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[179] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, pages 5449–5458, 2018.

[180] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=ryGs6iA5Km`.

[181] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Bowen Shi, Qi Tian, and Hongkai Xiong. Latency-aware differentiable neural architecture search. *arXiv preprint arXiv:2001.06392*, 2020.

[182] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018.

[183] Yongxin Yang, Irene Garcia Morillo, and Timothy M Hospedales. Deep neural decision trees. *arXiv preprint arXiv:1806.06988*, 2018.

[184] Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pages 40–48. JMLR. org, 2016.

[185] Jianbo Ye, Xin Lu, Zhe L. Lin, and James Z. Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *International Conference on Learning Representations (ICLR)*, 2018.

[186] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in neural information processing systems*, pages 4800–4810, 2018.

[187] Geng Yuan, Xiaolong Ma, Caiwen Ding, Sheng Lin, Tianyun Zhang, Zeinab S Jalali, Yilong Zhao, Li Jiang, Sucheta Soundarajan, and Yanzhi Wang. An ultra-efficient memristor-based dnn framework with structured weight pruning and quantization using admm. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.

[188] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *Proceedings of the British Machine Vision Conference (BMVC)*, 2016.

[189] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJe8pkHFwS`.

[190] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient CNN implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016.

[191] Chi Zhang, Yujun Cai, Guosheng Lin, and Chunhua Shen. Deepemd: Few-shot image classification with differentiable earth mover's distance and structured classifiers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12203–12213, 2020.

[192] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[193] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

[194] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018.

[195] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[196] Ting Zhang, Guo-Jun Qi, Bin Xiao, and Jingdong Wang. Interleaved group convolutions for deep neural networks. *CoRR*, abs/1707.02725, 2017. URL `http://arxiv.org/abs/1707.02725`.

[197] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[198] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. DNNBuilder: an automated tool for building high-performance dnn hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*, page 56. ACM, 2018.

[199] Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, and Xinyu Niu. Towards efficient convolutional neural network for domain-specific applications on FPGA. *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 147–1477, 2018.

[200] Ruizhe Zhao, Xinyu Niu, and Wayne Luk. Automatic optimising CNN with depthwise separable convolution on FPGA: (abstact only). In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018.

[201] Yiren Zhao, Xitong Gao, Robert Mullins, and Chengzhong Xu. Mayo: A framework for auto-generating hardware friendly deep neural networks. In *Proceedings of the 2nd International Workshop on Embedded and Mobile Deep Learning*, pages 25–30, 2018.

[202] Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. Focused quantization for sparse CNNs. In *Advances in Neural Information Processing Systems*, pages 5584–5593, 2019.

[203] Yiren Zhao, Xitong Gao, Xuan Guo, Junyi Liu, Erwei Wang, Robert Mullins, Peter YK Cheung, George Constantinides, and Cheng-Zhong Xu. Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 45–53. IEEE, 2019.

[204] Yiren Zhao, Duo Wang, Daniel Bates, Robert Mullins, Mateja Jamnik, and Pietro Lio. Learned low precision graph neural networks. *arXiv preprint arXiv:2009.09232*, 2020.

[205] Yiren Zhao, Duo Wang, Xitong Gao, Robert Mullins, Pietro Lio, and Mateja Jamnik. Probabilistic dual network architecture search on graphs. *arXiv preprint arXiv:2003.09676*, 2020.

[206] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2423–2432, 2018.

[207] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless CNNs with low-precision weights. In *International Conference on Learning Representations*, 2017.

[208] Hao Zhou, Jose M. Alvarez, and Fatih Porikli. Less is more: Towards compact CNNs. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision*

– *ECCV 2016*, pages 662–677, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

[209] Kaixiong Zhou, Qingquan Song, Xiao Huang, and Xia Hu. Auto-GNN: Neural architecture search of graph neural networks. *arXiv preprint arXiv:1909.03184*, 2019.

[210] Shuchang Zhou, Zekun Ni, et al. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

[211] Yanqi Zhou, Xuanyi Dong, Berkin Akin, Mingxing Tan, Daiyi Peng, Tianjian Meng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. Rethinking co-design of neural architectures and hardware accelerators. *arXiv preprint arXiv:2102.08619*, 2021.

[212] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *International Conference on Learning Representations (ICLR)*, 2017.

[213] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems (NIPS)*. 2018.

[214] Marinka Zitnik and Jure Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 2017.

[215] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *International Conference on Learning Representations (ICLR)*, 2017.

[216] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.