Accelerating Convolutional Neural Networks Through Enhanced Designs



Marcelo Gennari do Nascimento

Christ Church College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity Term 2022

Acknowledgements

I would like to thank my supervisor, Victor Prisacariu, for years of close mentorship and trust in my work. From guiding me through an internship during my undergraduate to fighting for my DPhil place and funding, his belief in my potential, often bigger than my belief in myself, has bootstrapped my academic and career achievements. I would also like to thank my co-supervisor Roger Fawcett, who not only secured the scholarship that I needed to make this thesis possible, but also provided invaluable insight into the hardware inner workings. Our meetings in Basingstoke, at the IEB, and eventually at the Old Parsonage were the birthplace of a significant portion of our research ideas. I could not have asked for better people to guide me through the first years of my career.

I am very thankful to my collaborators at (former) Omnitek, Intel, and Microsoft, for trusting in my work and giving me the freedom to explore my ideas. I am also very thankful to all my colleagues at the Active Vision Lab, specially to my cohort lab mates Henry, Theo, Michael, Wesley, and Ryan. All of them have had the same unique experience of doing this DPhil as me, and their support, friendship, and debugging expertise have saved me countless hours, and provided the emotional support needed to carry on with my work.

Thank you to all my friends, specially Ben and Georgiana, who have stayed next to me and supported me through happy and tough times. From spontaneous long walks to the city centre, to having a relaxing time playing boardgames, they have achieved the almost herculean task of indulging me for almost a decade, and for that I am extremely grateful.

Finally and most importantly, I want to thank my family, my siblings Lucas and Luiza, my stepparents Rita and Wilson, and specially my parents Célia and Domingos, who have sacrificed so much to support me through years of education. Their support and care have been felt across the pond throughout my 8 years at Oxford.

Abstract

Although convolutional neural network (CNN) models have revolutionized and pushed the boundaries of computer vision, they often suffer from high memory and power costs due to the substantial amount of computation performed. The goal of this thesis is to provide algorithms, heuristics, and insights into accelerating CNNs.

The main method explored in this thesis is that of quantization. Normally neural networks use the standard IEEE 32-bit floating point format. However, due to the significant redundancy of these models, often only a fourth of the bitwidth is necessary, and in many cases this can be pushed down to even 2 or 1 bit. We present three papers exploring this idea. Our first paper, DSConv, proposes a channel-wise block floating point (BFP) format for tensor manipulation, which allows the bulk of the tensor computation to be performed using the low-cost high-throughput integer multiplications. This is achieved by sharing an exponent value across a group of mantissas, allowing the model to still have a wide representation power whilst avoiding floating point operations. Our second paper, NUQ, proposes an algorithm to find the optimal bitwidth of each layer of a CNN. It is expected that some layers need higher precision than others, and to avoid bottlenecks on compression, it is necessary to find the lowest bitwidth for each layer without losing overall accuracy. Our third paper, HyperBFP, focuses on expanding DSConv to the case of performing training and backpropagation in low precision. Quantizing gradients is more challenging than quantizing weights and activations in general, since it requires higher precision to work. Our approach is to extend the channel-wise BFP format to two dimensions in order to avoid having to requantize transposed convolutions in the backpropagation step.

A downside to new quantization schemes is that they rely on specifically designed hardware, such as field programmable gate arrays (FPGAs) and ASICs in order to accelerate the neural network. However, a significant proportion of edge devices do not have support for these state-of-the-art compression methods. In our fourth paper, we explore algorithms to compress and accelerate CNN models to run specifically on commodity hardware and current neural network accelerators. We propose a pipeline in which three compression stages are performed interleaved by recovering the accuracy using knowledge distillation. We show that by keeping the accuracy always close to the original model through knowledge distillation, we are able to achieve more than 5× speedup on commonly used superresolution networks without suffering significant perception loss.

Contents

List of Figures **vi**

1 Introduction **1**
1.1 The Importance and Applications of Convolutional Neural Networks 1
1.2 The Compute Costs of Convolutional Neural Networks 3
1.3 Thesis Goal and Outline 4
1.4 Publications 7

2 Literature Review **8**
2.1 Neural Network Design 9
2.2 Quantization 11
2.3 Knowledge Distillation 18
2.4 Tensor Decomposition 19
2.5 Pruning 22
2.6 Conclusion 23

3 DSConv: Efficient Convolution Operator **24**
3.1 Context 24
3.2 The Search for a Number Format 25
3.3 Author Contribution 28
3.4 Publication: Gennari do Nascimento *et. al.* *"DSConv: Efficient Convolution Operator"* ICCV 2019 28
3.5 Conclusion 40

4 Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes **41**
4.1 Context 42
4.2 NAS Algorithms 42
4.3 Author Contribution 46
4.4 Publication: Gennari do Nascimento *et. al.* *"Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes"* ECCV 2020 46
4.5 Conclusion 64

5 HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation 65
5.1 Context . . . 66
5.2 A Primer on Backpropagation in Convolutional Layers . . . 66
5.3 Author Contribution . . . 68
5.4 Publication: Gennari do Nascimento *et. al. "HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation" In Prep.* . . . 68
5.5 Conclusion . . . 83

6 A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks 84
6.1 Context . . . 85
6.2 The usage of Neural Network Accelerators . . . 85
6.3 Author Contribution . . . 89
6.4 Publication: Gennari do Nascimento *et. al. "A Knowledge Distillation Approach toPruning and Accelerating SuperresolutionNeural Networks" In Prep.* . . . 89
6.5 Conclusion . . . 110

7 Discussion and Conclusion 111
7.1 Discussion . . . 112
7.2 Achievements . . . 112
7.3 Future Work . . . 114

References 117

List of Figures

1.1 Size, accuracy, and total floating point operations of common CNN architectures. The figure is reproduced from [27], which is an updated version of the original figure from [28]. 4

2.1 Density of accuracy per architecture size. The figure is reproduced from [27], which is an updated version of the original figure from [28]. 10
2.2 Energy expenditure for different operations at different levels of precision. Figure is taken from [48], which used energy numbers from [49]. The energy and area cost can be approximately modelled as increasing with the square of the number of bits for the same type of operations. 12
2.3 Visualization of the Straight Through Estimator (STE) [50]. 14
2.4 Pipeline used in the majority of Quantization-Aware Training (QAT) methods. 16
2.5 Separable Convolution being used to save computational complexity. This shows a 2D Gaussian Filter being decomposed into two 1D Gaussian Filter components: one horizontal and the other one vertical. Image taken from [81], page 406. 20
2.6 Popular ways of decomposing tensors. Image taken from [86]. 21

3.1 Different Floating-Point configurations for running neural networks. This includes the most common IEEE754 formats [47], such as double precision, single precision, and half precision. It also includes BFloat16 [107], TensorFloat-32 [108], used at NVidia A100 GPUs, and the two verisons of FP8, used at NVidia Hopper GPUs [109]. . 26
3.2 Difference between a Floating Point format and a Block Floating Point format. 27

5.1 Illustration of operations needed in order to perform backpropagation in convolutional layer. 67
5.2 Illustration of exponent misalignment caused when a block floating point tensor is transposed. Same colour indicates values that share the same exponent. 67

6.1 Time taken to run ResNet18 on ImageNet. 87
6.2 Channel Sampler for branch problem. Image taken from [104]. . . . 88
6.3 Energy consumption for common arithmetic and memory operations. Image taken from [144]. 88

1 Introduction

Contents

1.1 The Importance and Applications of Convolutional Neural Networks 1
1.2 The Compute Costs of Convolutional Neural Networks 3
1.3 Thesis Goal and Outline 4
1.4 Publications 7

1.1 The Importance and Applications of Convolutional Neural Networks

Since their inception and popularization, convolutional neural networks (CNNs) have revolutionized the field of Computer Vision. Because the models are so general, they can be used as backbones for the majority of the tasks that require pattern recognition in images. The idea of convolving an image using a kernel to extract features dates back to the early days of digital image processing. They were at first designed by hand: a simple early example of such algorithms can be traced to the Sobel Edge Detector [1], which was a discrete approximation of a differential operator; similarly, the Gaussian Filter [2] was a particularly important hand-designed filter and widely used throughout the history of Computer Vision,

being a part of complex feature extraction algorithms such as the Canny Edge Detector [3] and SIFT (Scale-invariant Feature Transforms) [4].

The advent of Convolutional Neural Networks and the backpropagation algorithm allowed researchers to stack multiple kernels one after the other and "learn" their values without the need to design them by hand. This proved to be a more successful approach to tasks such as detection and classification. The early work [5, 6] that started the popularization of CNNs achieved high accuracy in the MNIST (*Modified NIST*) dataset [7]. They were subsequently popularized with the introduction of the AlexNet model [8], which demonstrated that CNNs could scale with data by winning the ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*) [9, 10].

Since the inception of AlexNet, CNNs have been used to solve a variety of different tasks. Their applications span image classification [11–13], to object detection [14, 15], semantic segmentation [16], pose estimation [17], superresolution [18], among others. Their quality as general feature extractors also allowed researchers to use pretrained neural networks as a backbone to transfer their capabilities to other tasks, making it easier to apply these models to a plethora of computer vision tasks [19].

As a consequence of the success of these models, there was a need to provide more support for running neural networks in edge devices, especially mobile phones. Running neural networks directly in the device is important for three main reasons: i) it allows offline usage, and potentially decreases latency as data does not need to leave the device; ii) it alleviates the demand for servers; iii) it solves many security and privacy problems, since data collected in the edge (photos and videos for example) stay in the edge, and do not need to be sent to a centralized server in order to be processed for a specific application. Consequently, chip vendors such as Qualcomm and ARM have included dedicated hardware and software for running specifically applications backed by neural networks. For example, Qualcomm has released the Snapdragon Neural Processing Engine (SNPE) [20], and ARM has embedded in their chips a Neural Processing Unit (NPU) [21]. However, running these models on the edge can be challenging because of their high compute cost.

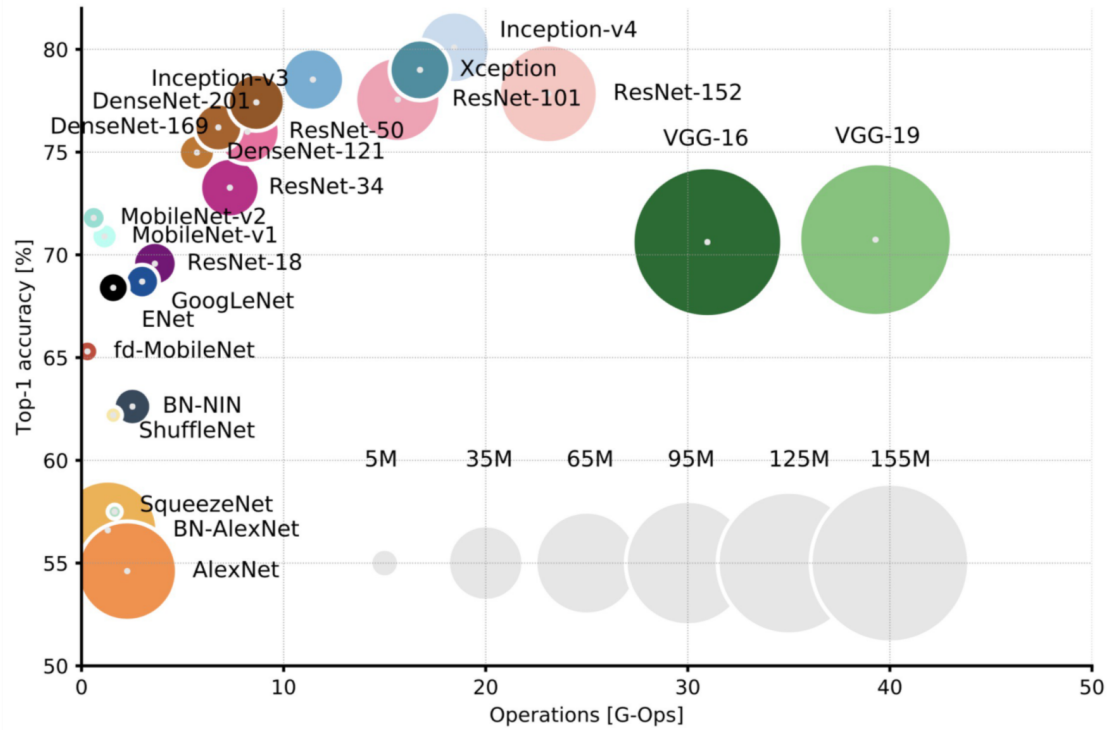
1.2 The Compute Costs of Convolutional Neural Networks

One of the biggest problems with CNNs is the fact that they require a large amount of computing power. This causes two problems: i) Data centres are overwhelmed with inference queries, which can incur significant energy and infrastructure costs; ii) Edge device owners, specially mobile phones, have strict memory and battery life requirements. Decreasing both memory usage and power usage is therefore a priority.

Figure 1.1 shows the most common neural network architectures by the number of parameters (which defines their memory storage usage), Giga Floating Point Operations (GFLOPs - which correlates their energy consumption), and accuracy. The early CNNs such as VGG and AlexNet had a large amount of parameters, in the order of 70 million. This number has been reduced dramatically, resulting in the creation of networks such as the MobileNet family [22–24], ShuffleNet [25], and SqueezeNet [26], all of which contain less than 10 million parameters. However, even the smallest networks still require number of operations in the order of GFLOPs. The energy requirement is even higher when considering the memory accesses and data transportation needed to run and train these networks.

CNNs are naturally redundant [29] because they are overparameterized models by design. The success of training CNNs has been directly associated with its overparameterization [29]. It is also argued that their redundancy makes the models more robust, since a failure in a set of latent variables might not affect the overall results of the inference [30]. Another consequence of overparameterization is that large amounts of its parameters can be removed without considerably reducing their accuarcy [31]. The last point is the main concern of this thesis. The fact that CNNs are highly compressible encourages the search for compression algorithms. These algorithms aim to reduce compute costs, and memory requirements without sacrificing accuracy.

Figure 1.1: Size, accuracy, and total floating point operations of common CNN architectures. The figure is reproduced from [27], which is an updated version of the original figure from [28].



1.3 Thesis Goal and Outline

This thesis' objective is to explore methods to design and optimize networks in order to produce lighter models, that run quicker on the target hardware without losing significant accuracy. It is organized as an integrated thesis, meaning that its content comes primarily from publications [1].

This thesis explores accelerating neural networks under two approaches: the first is that of writing software to run on specifically designed hardware, and the second is of designing algorithms to run on general-purpose hardware. The first approach elaborates on one of the main neural network compression methods - quantization, which leaves the architecture of the network intact and changes only its number representation to take advantage of specifically designed hardware. There are 3 papers included in this thesis that explore this topic, which are presented in

[1] https://www.mpls.ox.ac.uk/graduate-school/information-for-postgraduate-research-students/submitting-your-thesis

Chapters 3, 4, and 5; The second approach explores architectural changes of neural networks that exploit an existing target hardware for efficient inference. It includes the last paper, presented in Chapter 6, which aims to accelerate superresolution networks targeted at the Qualcomm Snapdragon [32] devices.

Chapter 2 provides a high level overview of the vast literature on accelerating neural networks. Its aim is to lay the foundation of the community efforts on compression, and give to the reader a succinct summary of methods to run CNNs using less memory and with higher throughput.

The first paper, presented in Chapter 3, is the basis of our quantization work and it is called *DSConv: Efficient Convolution Operator*. It was presented at the International Conference on Computer Vision (ICCV) in 2019. The proposition of the paper is to introduce a new quantization method for convolutional layers. Previous methods have either used a floating point representation (for example BFloat16 [33]), or a fixed point representation. We argue that both representations have their benefits: floating point has a better usage of the range of the possible values whereas fixed point saves the most silicon area. Our paper proposes using a representation that is a mix of both of these called Block Floating Point. Instead of using one exponent per value in the tensor, we instead share an exponent depthwise across a block of integer values. This has the benefit of solving the range problem and still being able to use 2s complement for the majority of the computation.

Similary to the majority of the previous literature, *DSConv* assumes that all of the layers are quantized to the same precision. However, intuitively some layers should require higher precision than others, which results in some layers not being quantized optimally. Our second paper (Chapter 4), *Finding Non-uniform Quantization Schemes Using Multi-task Gaussian Processes*, presented at the European Conference on Computer Vision in 2020, addresses this problem: we propose an algorithm to find the best distribution of bits through the layers of a CNN.

Our third paper (Chapter 5) expands the quantization approach from DSConv to the backpropagation case. The majority of previous methods focus on quantizing only inference. However, the training step requires more than twice the amount

of time needed to perform one iteration compared to the forward pass. For the neural network designer, having smaller cost of training models is of paramount importance. Applying DSConv naively to backpropagation causes problems though, particularly regarding the sharing of the exponent depthwise, since during backprop gradients are transposed, which makes the blocks misaligned. Our paper, *HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation*, which is submitted to the Winter Conference on Applications of Computer Vision (2023), addresses this problem by proposing quantizing the tensors in two dimensions: both channel-wise and batch-wise. This solves the misalignment problem since quantizing in both dimensions makes it invariant to transposition between those two dimensions.

Quantisation is an effective way of saving memory and compute power when it is run on purposely-built hardware. However, due to the cost of building new chips and shipping to the end user, the support for quantisation schemes is very limited to the general public and most companies, which often provide support to only naive 8-bit integer quantisation. For that reason, there is a necessity to explore methods to accelerate CNNs on hardware that already exists.

The introduction to Chapter 6 gives an overview on the types of support for compressed networks of current commodity hardware, the initiative of the research community on targeting these devices, and what methods have been tried to accelerate them. It also presents our fourth paper, *A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks*, which was undertaken whilst I was doing an internship at Microsoft. The main insight is that it gets increasingly more difficult to apply any compression method when the accuracy difference between the compressed network and the original network gets large. Therefore, applying multiple compression methods sequentially might quickly hinder the efficacy of each of them. We tackle this problem by interpolating knowledge distillation with a different compression methods (pruning, tensor decomposition and quantization), such that at each stage the compressed network is as close as possible to the original one.

The last chapter (Chapter 7) includes a discussion of the work done, and concludes the thesis gathering the main insights. It also includes many different lines of research to expand on the work that has been done in this thesis.

1.4 Publications

This thesis contains the following 4 papers:

- **Gennari do Nascimento, M.**, Fawcett, R., and Prisacariu, V. A. (2019). DSConv: Efficient Convolution Operator. In *International Conference on Computer Vision (ICCV)*
- **Gennari do Nascimento, M.**, and Prisacariu, V. A. (2020). Finding Non-Uniform Quantization Schemes Using Multi-Task Gaussian Processes. In *European Conference on Computer Vision (ECCV)*
- **Gennari do Nascimento, M.**, Fawcett, R., and Prisacariu, V. A. (2022). HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation. *Submitted for Publication*
- **Gennari do Nascimento, M.**, Vijayan, K., Sengupta, S., Neopythou, A., and Sommerlade, E. (2022). A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks. *In manuscript format.*

2
Literature Review

Contents

2.1 Neural Network Design 9
2.2 Quantization 11
2.3 Knowledge Distillation 18
2.4 Tensor Decomposition 19
2.5 Pruning 22
2.6 Conclusion 23

Since the early days of the development of neural networks, the problem of reducing computational costs has been extensively researched. Due to the recent wide adoption of CNNs, there has been a surge of interest from both academia and industry to reduce the cost of running neural networks.

This chapter will cover the literature of five related topics. The first one is Neural Network Design, which aims to search for more efficient architectures by creating more efficient network topologies. The second topic is Quantization, which is a technique to accelerate the network by changing its number representation to a more compact format. For example, a network that uses 32 bits for each value could use 8 bits, which incurs significant savings in both memory and power. The third topic explored is that of Knowledge Distillation, which aims to increase the accuracy of a network by using a bigger, more accurate network as a teacher,

without making any modifications to the two networks at all. The fourth topic is Tensor Decomposition, which seeks to apply some well known linear algebra manipulations (such as separable matrices) to tensors, and thus reduce the total number of operations needed. The fifth and last topic explored is Pruning, which aims to accelerate the network by sparsifying its representation. This chapter then ends with a conclusion summarising what was learnt so far.

2.1 Neural Network Design

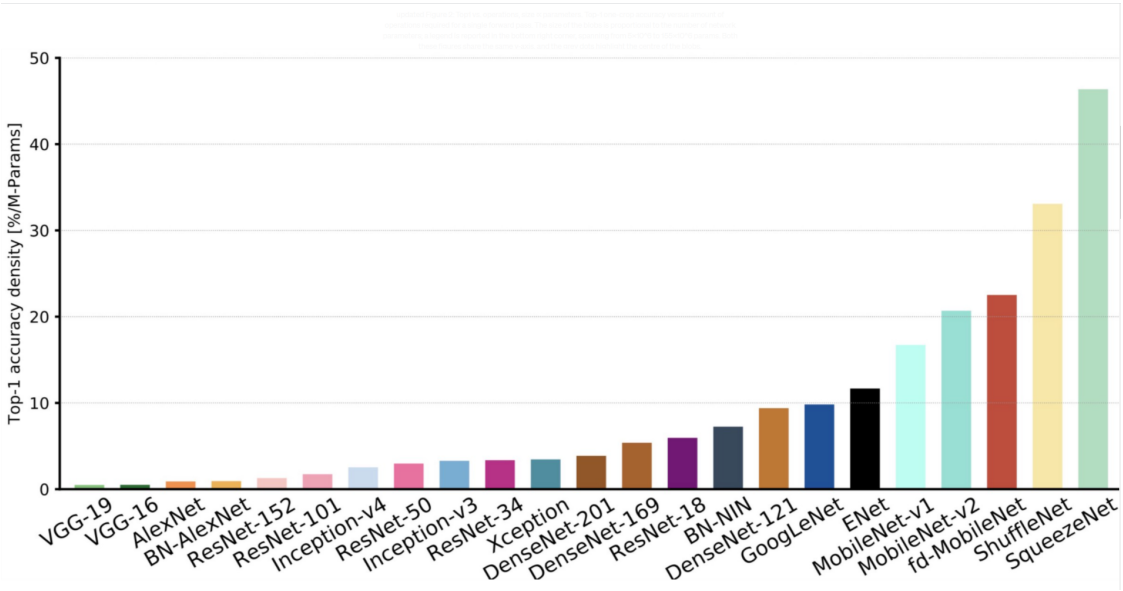
One of the primary focuses of the community has been in designing better architectures, which can achieve both higher accuracy, take less time to run, and use a lower number of operations. This effort has resulted in a number of insights on best practices for neural network design.

Initially, researchers hand designed efficient architectures, using heuristics to guide whether a certain structure would be more or less compact with the same representation power. Naturally, architectures such as the VGG [12] and ResNet [11] families achieved better performance, and were more efficient when compared to AlexNet [8]. Early works focused on compression and explored the use of different kernels and factorizations of the weights [34, 35]. This effort culminated in a substantial improvement in efficiency. Among the many architectures that popularized more efficient architectures is SqueezeNet [26], which successfully employed the strategy of using 1x1 filters, and MobileNet [22], which introduced the idea of depth-wise convolutions, and could achieve better results than ResNet-18 whilst using a fraction of its GFLOPs and parameters. Other popular architecutres that built on the main insights from the papers above include DenseNet [36], which employs heavy usage of residual connections, and ShuffleNet [25], which tries to increase representation by permuting channels in the feature maps.

Figure 2.1 illustrates the progression of these efforts. The plot shows the accuracy divided by the number of parameters (essentially a measure of density of accuracy) for different types of architectures. In the most extreme case, the improvement

from VGG-19 to SqueezeNet is more than 100 fold. It is also important to note that SqueezeNet achieves higher accuracy than AlexNet.

Figure 2.1: Density of accuracy per architecture size. The figure is reproduced from [27], which is an updated version of the original figure from [28].



The main area of research which tries to find optimal architectures automatically, is called *Neural Architecture Search* (NAS). The paper that popularised the automatic search of architectures was [37], which used a reinforcement learning algorithm to train an RNN to automatically create architectures. Notably, this approach was far too costly: to find an architecture in the CIFAR10 dataset [38], they used approximately 22,400 GPU-hours (800 GPUs running for 28 days). Since then, this cost has been reduced significantly, first with NASNet [37], which introduced two key ideas: i) searching cells instead of full architectures; ii) searching on a smaller proxy dataset and transferring to a larger dataset. They were then able to reduce the computational load to 2,000 GPU-hours (500 GPUs for 4 days). Another popular approach has been to find a subgraph of a large network by either using gradient information such as in DARTS [39], or by sharing parameters across nodes such as in ENAS [40]. Since then there has been an explosion of papers attempting to reduce the computational cost of NAS, whilst adhering to specific computational load and accuracy restrictions [41–44]. An insight on finding the architecture automatically

is that the search space can be defined for a specific hardware. This *hardware-aware* optimization relies on feedback from running the newly found architecture on the hardware and using its time and energy signals as a guide to where to search next. Papers that have successfully employed this method include MNasNet [45] and FBNets [46], which use feedback from a mobile device to find optimal architectures.

This idea of NAS is particularly important for Section 4, and therefore we have postponed some of the discussions on different search algorithms for NAS to the introduction of that section. There are some drawbacks to Neural Architecture Search: i) it requires a large amount of compute, since the search space of the possible neural networks is unbounded, and often each of these networks have to be trained; ii) it does not solve the problem of making existing pretrained networks run faster: the user would have to retrain the found network on the same data, which incurs development cost and time, especially for large networks. Other works have developed ways of compressing neural networks that can address these problems.

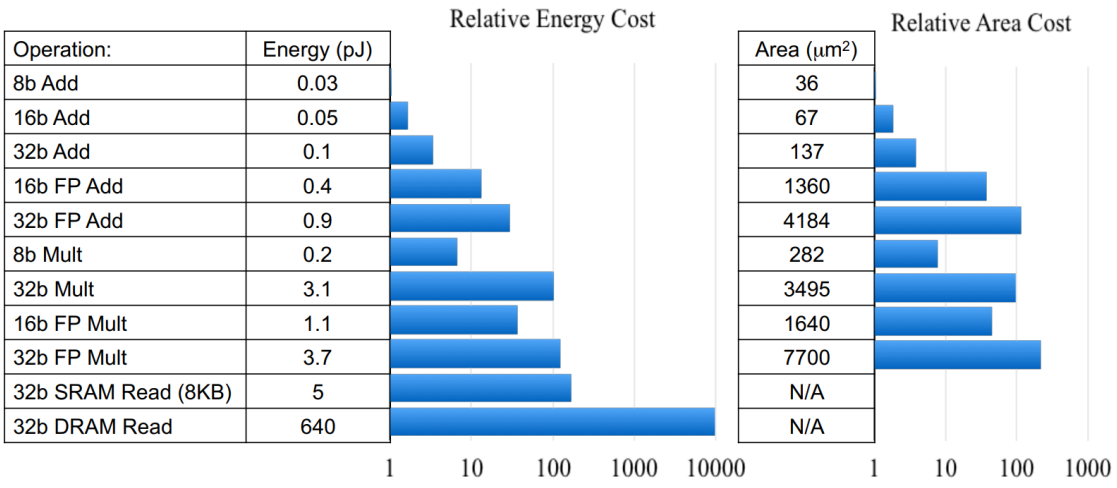
2.2 Quantization

One of the most popular ways of saving memory and power is to quantize the parameters and activations of the neural network. There are different definitions for what quantization means. In this thesis, the term is going to be used to describe that at least part of the tensors in the neural network is represented in a format that uses less bits than the standard IEEE754 Single Precision Floating Point [47], which requires 32 bits per number: 1 for the sign, 8 for the exponent, and 23 for the mantissa.

The main benefit of using less bits per number is that hardware can be designed to take advantage of the lower precision. The lower the precision, the faster and more power efficient the hardware can be designed for that application. Figure 2.2 shows the energy and silicon area requirements for different operations under different levels of precision. As can be seen, the silicon size required to run an operation with n bits can be approximately modelled as growing quadratically with n. Therefore reducing the bitwidth from 32 to 16 for example should reduce the

silicon area by 4 times, and a reduction from 32 to 8 bits should reduce the silicon area by 16 times (in the figure, 32b to 16b FP Mult goes from 7700μm^2 to 1640μm^2, a ratio of 4.7, and 32b to 8b Mult goes from 3495μm^2 to 282μm^2, a ratio of 12.4), which consequently also reduces the power usage by that same amount (in the figure, 32b to 16b FP Mult goes from 3.7pJ to 1.1pJ, a ratio of 3.4, and 32b to 8b Mult goes from 3.1pJ to 0.2pJ, a ratio of 15.5). It is also important to note the discrepancy on the usage of number formats: integer multiplication uses slightly less energy and silicon area than floating point multiplication (3.7pJ compared to 3.1pJ), and integer addition uses significantly less energy than floating point addition (0.9pJ to 0.1pJ, a difference of 9-fold). Lastly, it is also important to take into account the effect that memory access takes into the energy equation: the difference between DRAM and SRAM expenditure is more than 100-fold, which emphasises the need to have small tensors which fit in cache.

Figure 2.2: Energy expenditure for different operations at different levels of precision. Figure is taken from [48], which used energy numbers from [49]. The energy and area cost can be approximately modelled as increasing with the square of the number of bits for the same type of operations.



Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Reducing the precision of the neural network can cause some problems. Generally, it is expected that the CNN will lose accuracy with lower precision, since its parameters are now restrained to a certain set of values. It is the job of the

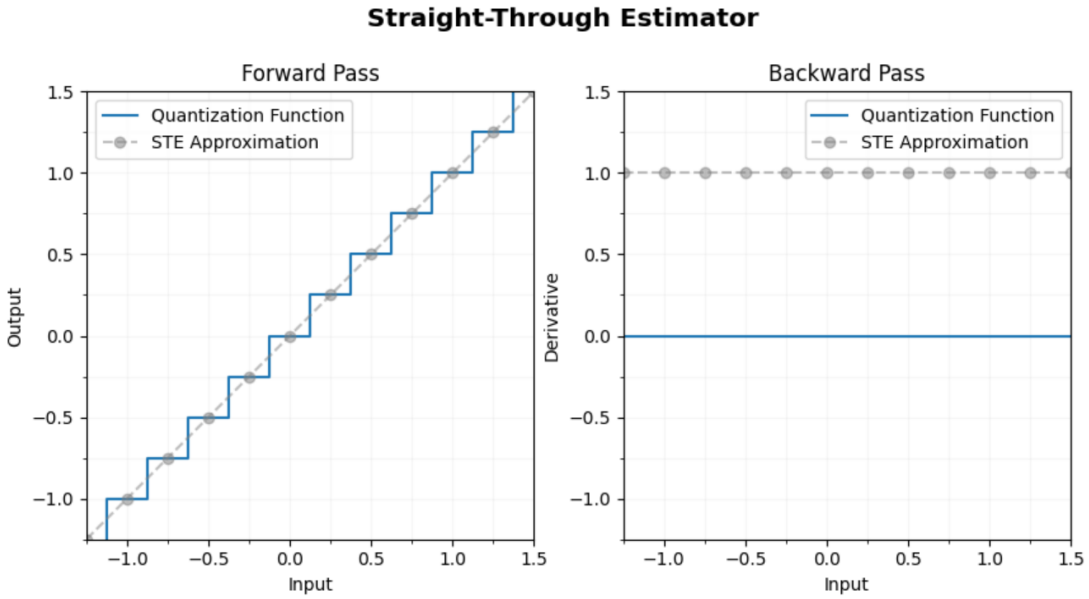
designer to find the trade-off between performance and accuracy that satisfies the task they are trying to solve.

The quantization procedure can be done for different motivations. Most of them however stumble upon the problem of differentiability: in order to train the weights of a neural network, every node has to be differentiable (or at least allow for subgradients to be calculated) in order to allow backpropagation to be used. However, the quantization operator has derivatives equal to zero everywhere (with the exception of the boundaries, in which they are equal to infinity). This means that all gradients backpropagated through this operator would be equal to zero, and no weights would be updated. Most of the methods solve this problem in a similar fashion, by using Straight-Through Estimator (STE) [50], which is depicted in Figure 2.3. It simply consists of assuming that the forward pass is actually an identity function, and approximates its gradient as 1. From the figure, in the forward pass the Quantisation Function is used (in this case it is a rounding operator, where all input values are rounded to the nearest 0.25), which has derivatives of 0 everywhere, ignoring the step changes. In the backward pass however, the approximation to gradients equal to 1 is used instead, which allows the gradients to backpropagate without problems. It is a simple and biased estimator of the quantisation function, however it has been empirically shown to work exceedingly well [51].

The literature on quantization of neural networks is vast, and many different ways of categorizing the different lines of research have been proposed [52]. In this thesis we will break it up into three main goals: quantizing for storage, inference, and training.

Quantizing for Storage The simplest way to quantize a neural network is by quantizing its parameters only, whilst keeping activations and gradients in FP32 format. Due to the fact that only the weight of the neural network is being compressed, the accuracy drop caused by this quantisation is generally small, allowing for a large compression ratio to be achieved. Also the quantization scheme used does not need to obey hardware constraints, which generally assume that numbers are linearly (in the case of signed integer or 2s complement format)

Figure 2.3: Visualization of the Straight Through Estimator (STE) [50].



or loglinearly spaced (in the case of floating point format). This means that hashing and other techniques can be used to further increase the flexibility of the quantization process.

The main work on this is Deep Compression [53], which was able to reduce storage requirements of some popular networks such as AlexNet [8] by 35× to 49× without any accuracy drop, using quantisation and weight sharing as part of their pipeline. Weight sharing in similar fashion has also been used in HashNet [54]. Other methods also include BinaryNet [55] which was the first paper to have quantised only the weights of AlexNet to a single bit, but resulted in a significant drop of 21.2% in Top1 Accuracy. This was followed by XNOR-Net [56], which achieved no accuracy drop when quantizing the weights to 1 bit. Many more papers followed, including Ternary Weight networks [57], Trained Ternary Quantization [58], and LQ-Nets [59], which quantized more challenging architectures such as ResNets [11], and GoogLeNets [13].

Although this task allows for quantization schemes of down to 1 bit to be found, the usefulness of this task is limited: one of the main constraints of neural networks is the computational load and inference time that it requires. There are only two

very limited benefits in quantising only the weights of the neural network during inference time: i) it can help accelerate computation by decreasing memory access and cache allocation; ii) in the case of binary weights, all of the floating point operations can be reduced to summations only, without the usage of multiplications. Apart from those two reasons, quantizing only the weights of a neural network is only good at reducing the size of the neural network for storage, which is a problem that is not of utmost importance: ResNet18 [11] for example requires 45MB, whilst more modern architectures such as MobileNetV3 [60] can get to 21MB. A more useful motivation is to quantize both weights and activations to allow low precision MAC operations to be performed.

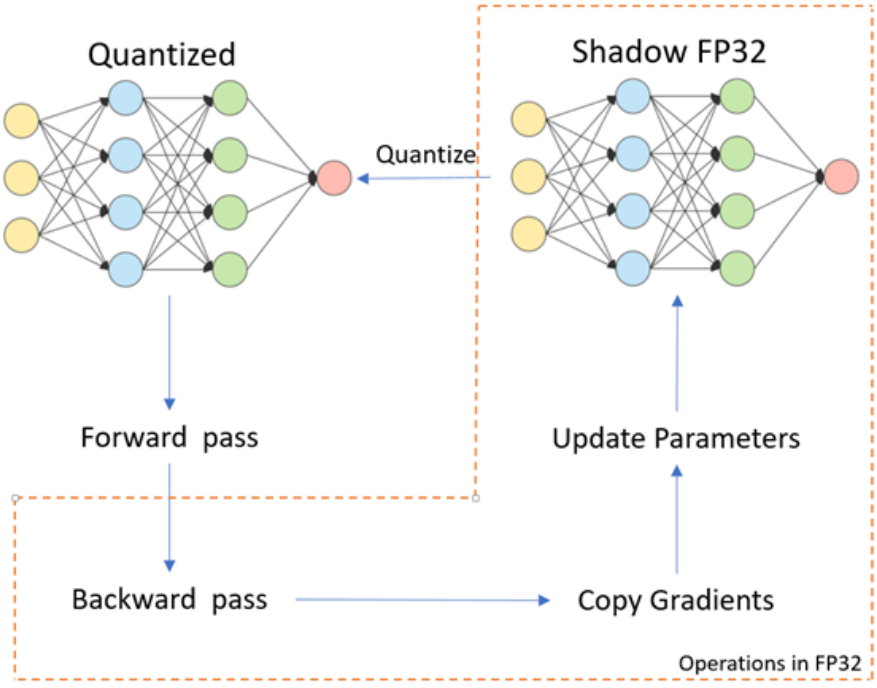
Quantizing for Inference One of the main reasons for using quantization is to accelerate the computation of the forward pass, improving latency and throughput. This means that both the weights and the activations need to be quantized, which allows for specially built hardware to multiply and accumulate values in low precision. There are two ways to quantize for inference: i) Quantize without the use of backpropagation, often with limited or no data available, which is also known as "post-training quantization" (PTQ); ii) Quantize with the use of backpropagation, often with original dataset available, which is also known as "quantization-aware training" (QAT).

PTQ is particularly important when the optimizer does not have access to the data for privacy policy reasons for example, or when training is prohibitively expensive. In this case, after quantization, there are limited options to recover the degradation in accuracy. There has been a surge of research in this area: quite a few papers suggested the use of a small set of calibration data used to only estimate the distribution of activations in hidden layers such that they fit within a certain range [61, 62]; the work of [63] also introduces the idea of using a small amount of calibration data to determine whether FP values should be rounded up or down. A significant effort lately has been also put on quantizing networks without the need of data at all. The strategy used to achieve this is often based

on correcting the values of parameters based on an error metric [64, 65]. Other approaches have also been tried, such as using a generative method to use synthetic data before calibrating the neural network [66–69].

Since it is expected that the quantization process will result in accuracy drops, a considerable amount of the drop can be recovered by fine-tuning or retraining the network under the assumption that it is quantized. QAT allows for the best results in accuracy, however it does require access to the data, and time for computation. Most papers that use QAT follow the pipeline outlined in Figure 2.4. The original neural network becomes a "shadow" network, which keeps all of its values in FP32. Its weights are then quantized using STE, and then the forward pass is performed (if activations are quantized, then STE is also applied to them). The backward pass is performed in FP32, because it usually requires higher granularity in order to train the CNN properly. The result of the backpropagation is the gradient with respect to each of the parameters of the CNN, which are updated all in FP32.

Figure 2.4: Pipeline used in the majority of Quantization-Aware Training (QAT) methods.



Many papers have successfully used this model to train a quantized neural

network, such as LQNets [59], ABC-Nets [70], XNOR-Net [56], DoReFaNet [71], and HWGQ [72].

Quantizing for Training The ideal scenario is when the network is quantized in both forward and backward passes without loss of accuracy. Since the backward pass requires roughly twice as much computation, accelerating that step is of critical importance for engineers and neural network developers. This is also the most restricted and less understood scenario, because not only are the parameters constrained to particular values, but also the gradients are constrained as well.

A common issue when quantizing for training is that gradients become closer and closer to zero during optimization, which means that they get rounded to zero very often. This causes problems such as *stagnation* (more on that on Chapter 5), as gradients are all zero and do not update the weights as intended. A workaround that the field has found is to use *stochastic rounding* [73–75] instead of using round-to-nearest for gradients.

There are many different ways that rounding can be performed. Deterministic rounding is usually split into many modes, the most used ones being the *ceiling* operations (notated as $\lceil x \rceil$), which rounds up, the *floor* operation (notated as $\lfloor x \rfloor$), which rounds down, and round-to-nearest, notated as $\lfloor x \rceil$. In the round-to-nearest method, the value 0.5 is ambiguous, since it can be either rounded down or up. So there are variants for tie breaking rules: *round half towards zero* rounds ambiguous values towards zero, which means that positive values get the ties rounded down (1.5 to 1.0 for example) and negative values get the ties rounded up (-1.5 to -1.0) for example. Similarly *round half away from zero* rounds positive ties up (1.5 to 2.0) and negative values down (-1.5 to -2.0). Analogously there is also the possibility of *rounding to even* (1.5 goes up to 2.0, but 2.5 goes down to 2.0) and *rounding to odd* (1.5 goes down to 1.0, but 2.5 goes up to 3.0). These tie breaking rules avoid generating biases in the rounding operations, and they are used differently in different fields. IEEE754 [47] established 4 rounding modes, the default being rounding to nearest with ties rounding towards even numbers, which means that

either for positive or negative values, ties get rounded up and down 50% of the time each. For neural networks, even though round-to-nearest yields a network that is closest to the original network in the weight-space, it does not necessarily correspond to the one closest to the original network in accuracy, which has been demonstrated by the work of Nagel et.al. [63].

An alternative to the common deterministic rounding operations is what is called *stochastic* rounding. In this mode, rounding the same number can yield different results. There are two main modes for stochastic rounding: i) the probability can be linearly weighted based on how far that number is from the rounded number (for example, 0.8 would have 80% chance of being rounded up); ii) the probability of being rounded up or down is 50%.

There are a number of papers that have attempted to train using quantized gradients. The most important ones are DoReFa-Net [71], which uses the concept of stochastic rounding in order to quantize gradients, and establishes that it is necessary to make low bitwidth gradients effective. Another work that is able to successfully quantize gradients is SWALP [76], which employs averaging of different gradients in order to suppress errors in caused by the gradient quantization.

Quantization has a couple of limitations though, the most important one being that it only works in hardware that supports it. In commodity hardware, the support for 8-bit integer operations has become more ubiquitous, however it is still rare to find support for 4-bit, 2-bit, or any support for training at all. These more advanced and lower bitwidth methods can therefore only be applied to custom made hardware, such as FPGAs or ASICs. To accelerate a neural network for commodity hardware, there are other options worth investigating, such as Knowledge Distillation, Tensor Decomposition, and Pruning.

2.3 Knowledge Distillation

It is expected that larger networks, even from the same family, such as ResNets, are able to achieve higher accuracy than their smaller and shallower counterparts.

This raises the question on whether we can use the knowledge acquired in the bigger networks to train smaller ones.

For classification problems, the task of learning relies on hard labels: *e.g.* for ImageNet, the target labels are 0 everywhere apart from one index, in which it is equal to 1. Although this feedback to the CNN is correct, it is not as informative as it could be. For example, for an image of a car, it would give a feedback of 0 for both cats and trucks, even though trucks are significantly more likely to be mis-classified in the car picture than a cat.

The idea behind Knowledge Distillation [77] is instead of using only the label to train a network, we also use the feedback from a bigger network. In this way, we will get a combination of soft targets (which likely will provide more informative feedback to the network being trained) and the label itself.

The general idea is to have a *teacher* network, usually a bigger version of a network from the same family, such as ResNet50, and a *student* network, usually a small version of a network such as ResNet18. The student then learns by using a combination of the labels and the output of the teacher network.

This line of research has yielded a variety of methods for compressing neural networks. The work of [78] for example introduced the concept of using multiple intermediate teachers to bridge the potentially too big computational gap between the teacher and the student network. It has also inspired the idea of self-distillation [79], in which deeper portions of the network are used to "teach" shallower portions of the same network instead of using two distinct ones. This method has been widely used also in combination with quantization [80].

2.4 Tensor Decomposition

Another way to make architectures run faster is instead of making the operations run faster, create architectures with fewer operations. In Linear Algebra and Image Processing, the idea of Separable Convolutions [81] has been extensively used to save computation.

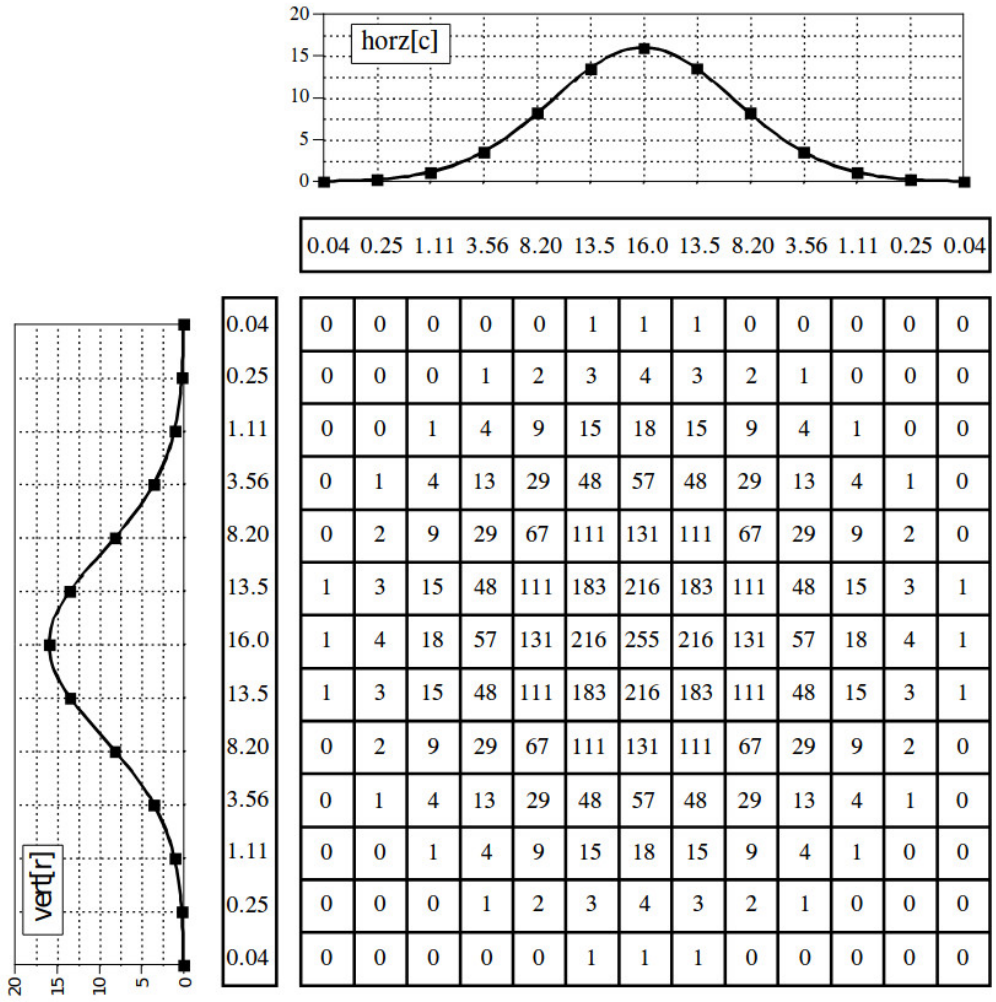


Figure 2.5: Separable Convolution being used to save computational complexity. This shows a 2D Gaussian Filter being decomposed into two 1D Gaussian Filter components: one horizontal and the other one vertical. Image taken from [81], page 406.

Figure 2.5 shows how the method works. Given a matrix $A_{n\times n}$ and a vector $\boldsymbol{x} \in \mathbb{R}^n$, the multiplication $A_{n\times n}\boldsymbol{x}$ takes $\mathcal{O}(n^2)$ multiplication operations. If $A_{n\times n}$ is separable though, in other words, if it can be written in the form of $A_{n\times n} = B_{n\times 1}C_{1\times n}$, then a large computational saving can be achieved: $A_{n\times n}\boldsymbol{x} = B_{n\times 1}C_{1\times n}\boldsymbol{x}$, and the complexity falls to $\mathcal{O}(2n)$ multiplication operations. In the example in Figure 2.5, the number of Floating Point Operations (FLOP) for the original matrix is 169. Breaking it down to two filters reduces it to 26. Not all matrices are easily separable as in the example given (Multidimensional Gaussians are a notable exception). Previous methods have for example tried to learn separable filters [82] to reduce computational cost, in that case created as a general framework

used in filter banks. Many methods exist that approximate a matrix into lower dimension ones in order to save computational complexity. One of the most notable methods is Singular Value Decomposition (SVD) [83], which was proven to be able to find a solution for any Real Matrix.

This idea of decomposing matrices to an equivalent formulation in order to save computational complexity can be applied to higher order matrices as well (tensors), and specifically to convolutional neural networks. The idea of Depth-Wise (DW), Point-Wise (PW), and Group Convolution used in MobileNet [22], XCeption [84], and ShuffleNet [25], is now well known. It is also known that any normal convolution can be decomposed to a sum of DW+PW convolutions [85] in closed form.

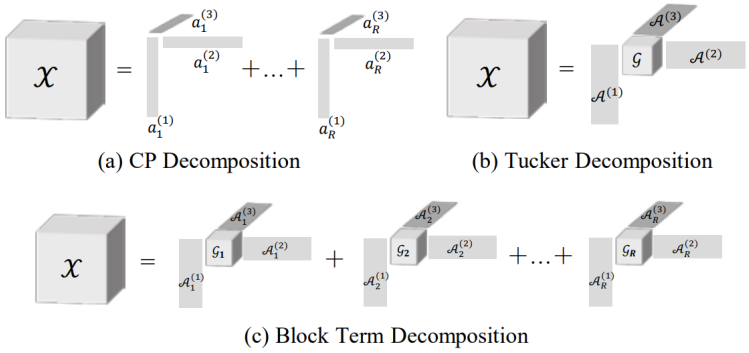


Figure 2.6: Popular ways of decomposing tensors. Image taken from [86].

Most times, this decomposition does not need to be exact. The idea of Low-Rank Approximation (Rank Pruning) is to decompose a tensor such that the decomposed one has lower rank than the original whilst still retaining most of its information. This is done by setting the lowest singular values to zero (pruning), such that the dimension of the rank can be ignored during computation. This has many implications including saving memory, computational complexity, dimensionality reduction, and others. One of the most popular methods in this segment is Principal Component Analysis (PCA) [87], which is used in many applications, such as the ones mentioned above. This topic has been extensively explored in application to Convolutional Neural Networks [86, 88–92].

Figure 2.6 shows some popular ways of decomposing tensors. Diagram (a) shows the **CP Decomposition**, which decomposes the tensor into sums of several rank-1 tensors. Diagram (b) shows the **Tucker Decomposition**, which decomposes the tensor into a core tensor and three tensor matrices, which is called a *tucker*. Diagram (c) shows the **Block Term Decomposition BTD**, which decomposes a tensor into a sum of *tuckers*. Additionally, based on the work of [86], there is a BTD decomposition tool that can be readily used.

2.5 Pruning

Another popular way of attempting to accelerate neural networks is by pruning - removing unnecessary values completely from the computational graph. We follow [93] and categorize pruning into two main groups: i) sparse-inducing algorithms; ii) non-sparse inducing algorithms.

Sparse Inducing Algorithms In this setting, individual weights (or activations) are set to zero, and the tensor representations become sparse. Usually, matrix multiplications are implemented in the naive way with computational complexity of $\mathcal{O}(n^3)$, where n is the size of the matrix. There are ways of accelerating this by using for example Strassen's Algorithm [94], which brings down the computational complexity to $\mathcal{O}(n^{log_2 7}) \approx \mathcal{O}(n^{2.807})$. These sub-cubic algorithms are generally better asymptotically, but not necessarily faster when moderate-sized matrices are being multiplied due to their initial overhead [95]. In practice, it is more productive to create algorithms that take advantage of the efficient usage of cache memory, by using optimization techniques such as blocking [96] to minimize the number of cache misses. However, by assuming sparsity and representing the matrix either in Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR), the computational complexity can dramatically fall. Unfortunately, this is not supported in most software packages or efficiently implemented in most hardware, and its benefits are not dramatic when using parallelism.

One of the early papers on this type of pruning is the seminal "Optimal Brain Damage" [97], in which individual weights of a neural network are pruned. Some other methods also involve unstructured pruning, such as the work of [98, 99], which introduced and popularized the idea of the pruning and retraining loop, and the work of "Lottery Tickets Hypothesis" and its consequences [100–102], which show that sparse neural networks can be trained from scratch, and surprisingly, a frozen neural network can achieve good results on small datasets by simply pruning its weights. Sparse inducing pruning can also be achieved by structured pruning, such as kernel pruning [93, 103].

Non-Sparse Inducing Algorithms The second way of pruning involves keeping the tensor representation as dense tensors. This is achieved by pruning entire blocks of channels or filters such that the resulting tensor is a smaller, dense version of the original. This method achieves lower sparsity levels than sparsifying individual weights and activations, but its advantage is that it can be directly used in current commodity hardware without the need of any special implementations. Previous papers have achieved this by pruning entire layers and channel blocks for example [104–106].

2.6 Conclusion

This literature review should provide to the user the main techniques explored by the community in the last couple of years on how to compress neural networks for efficient computation. We have provided an overview of the main ideas such as neural architecture search, quantization, knowledge distillation, tensor decomposition, and pruning. Each one of these techniques have specific applications and situations in which they thrive, as well as requirements for effective usage. The next chapters are going to introduce new ideas and algorithms explored during the DPhil which expand on this literature.

3

DSConv: Efficient Convolution Operator

Contents

3.1 Context 24
3.2 The Search for a Number Format 25
3.3 Author Contribution 28
3.4 Publication: Gennari do Nascimento *et. al. "DSConv: Efficient Convolution Operator"* ICCV 2019 28
3.5 Conclusion 40

In this chapter, we will mainly present our paper "DSConv: Eficient Convolution Operator", published at the International Conference on Computer Vision, in 2019. First we will start by giving some context on the motivation behind our work. Then we will provide some background information on number formats, and the different number formats explored by both the academia and the industry. We will then present our paper, and conclude with an overview of our findings and contributions to the community.

3.1 Context

CNNs are highly compressible, as it was argued in Chapter 1. One of the main methods for compressing them is quantization. The aim is to find a way of performing inference on any CNN topology by using fixed point integer operations only, which

has been shown in Figure 2.2 to require significantly lower power usage than floating point operations. Since the values of the weights are known and static, it is relatively easy to find appropriate scale values and thresholds to quantize weights. The main concern is that features maps are dynamic, since every image will yield different activation values. Therefore it is more difficult to find thresholds for quantization. This is a problem that drove researchers to investigate different number formats and precisions for neural network inference and training.

3.2 The Search for a Number Format

The number format of the neural network matters, because they dictate energy expenditure and accuracy of results. There are mainly two types of number representations: fixed-point numbers and floating-point numbers.

Fixed-point numbers are generally expressed in one of the following two ways: i) Signed integer format, which uses one bit for the sign, and the other bits for the magnitude - *i.e.* 1011 is -3 in binary. The main disadvantage is that the value 0 is duplicated - it can be expressed as negative 0 or positive 0 - and it requires different hardware implementations for addition and subtraction; ii) 2s complement, which converts positive values to negative by flipping their bits and adding one - *i.e.* 00101 is 5, and 11011 is -5. The advantages are that there are no duplicate 0s and subtraction can be performed with the same silicon as addition. Their operations are very efficient to be implemented in hardware. The main drawback with fixed-point numbers is that they are limited in their scale: since they are linearly distributed across the real number line, they can only express precision at a certain order of magnitude. It is not possible for them to be precise in the 100s, 1s and decimals for example, unless a significant number of bits is used. When quantizing neural networks, this can cause the problem of suppressing numbers to zero. For example, using the appropriate scaling factors, any value between -0.5 and 0.5 would be rounded to 0, which sparsifies the neural network and can cause stagnation during training since no information is transmitted when the tensors are multiplied by 0.

Floating-point numbers are widely expressed with three different parts: a sign bit, an exponent portion, and a mantissa portion. Let s be the sign bit, $e_0, e_1, e_2, ..., e_k$ be the exponent bits, and $m_0, m_1, m_2, ...m_n$ be the mantissa bits, resulting in a floating-point number with $1 + k + n$ total number of bits. Its value will be equal to $(-1)^s \cdot 2^{(e_0 e_1 e_2 ... e_k)_2 - 2^{k-1}+1} \cdot (1.m_0 m_1 m_2 ... m_n)_2$, where $()_2$ means that the number is in base-2. This format allows for the designer to trade-off the precision of the number via its mantissa bits, and the order of magnitude, via its exponent bits.

Figure 3.1: Different Floating-Point configurations for running neural networks. This includes the most common IEEE754 formats [47], such as double precision, single precision, and half precision. It also includes BFloat16 [107], TensorFloat-32 [108], used at NVidia A100 GPUs, and the two verisons of FP8, used at NVidia Hopper GPUs [109].



Naturally, both academia and industry have explored a large variety of floating-point number configurations. Figure 3.1 shows a small selection of the configurations

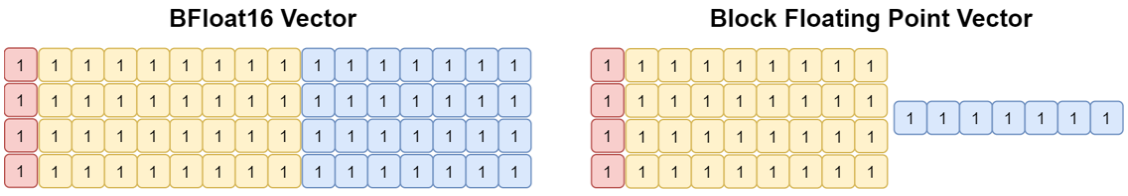
that has arisen from academic papers and chip manufactures. The main differences between the number formats represented in the figure are the number of bits allocated to either the mantissa or exponent portions of the representation.

There is an alternative to these two number representations: the *block floating point* (BFP) format, as illustrated in Figure 3.2. The BFP is only defined for groups of numbers, and it consists of sharing the value of one exponent across a block of numbers. There are a few benefits when using this format: i) when performing dot product between vectors, the great majority of the operations can be done using fixed-point arithmetic; ii) it can still represent numbers in different order of magnitudes as long as they are in different blocks with different exponent values.

Quantizing a block of values from floating point numbers to block floating points is also relatively simple and inexpensive: it consists of selecting the largest exponent across a group of values, and shifting all of numbers in the group such that they share the same exponent. Lastly, the numbers are truncated to the desired number of mantissa bits.

This allows us to dynamically quantize the features maps of a neural network without the need to use a calibration set, or guess the threshold values. We propose using the *block floating point* format channel-wise, since the dot-products will mostly be performed in this dimension. Among some of the benefits of this techniques is the fact that it is possible to keep the property of a large range of values under this representation whilst still using only fixed point arithmetic. Also, for a non-zero group of values, there will always be at least one activation value that will not be zero after quantization, which avoids the problem of suppressing feature maps to zero.

Figure 3.2: Difference between a Floating Point format and a Block Floating Point format.



Using this technique, we are able to quantize in many different tasks: we successfully quantized for storage (weights only), for quick inference without the usage of data, with unlabelled data by using knowledge distillation, and by retraining. We extensively test our method on classification networks, such as ResNet, GoogLeNet, and VGGNet, and datasets such as CIFAR10 and ImageNet.

3.3 Author Contribution

- A method to quantize neural networks by grouping values channel-wise and sharing exponents across them
- An algorithm that allows for integer multiply and accumulate operations without losing the range of floating point values
- Thorough experiments in many quantization settings, such as the cases with no data, unlabelled data, and fully supervised

3.4 Publication: Gennari do Nascimento *et. al.* *"DSConv: Efficient Convolution Operator"* ICCV 2019

DSConv: Efficient Convolution Operator

Marcelo Gennari do Nascimento
University of Oxford
Active Vision Lab
marcelo@robots.ox.ac.uk

Roger Fawcett
Intel Corporation
https://www.omnitek.tv/about
roger.fawcett@intel.com

Victor Adrian Prisacariu
University of Oxford
Active Vision Lab
victor@robots.ox.ac.uk

Abstract

Quantization is a popular way of increasing the speed and lowering the memory usage of Convolution Neural Networks (CNNs). When labelled training data is available, network weights and activations have successfully been quantized down to 1-bit. The same cannot be said about the scenario when labelled training data is not available, e.g. *when quantizing a pre-trained model, where current approaches show, at best, no loss of accuracy at 8-bit quantizations.*

We introduce DSConv, a flexible quantized convolution operator that replaces single-precision operations with their far less expensive integer counterparts, while maintaining the probability distributions over both the kernel weights and the outputs. We test our model as a plug-and-play replacement for standard convolution on most popular neural network architectures, ResNet, DenseNet, GoogLeNet, AlexNet and VGG-Net and demonstrate state-of-the-art results, with less than 1% loss of accuracy, without retraining, using only 4-bit quantization. We also show how a distillation-based adaptation stage with unlabelled data can improve results even further.

1. Introduction

A popular method to make neural networks faster and use less memory is quantization, which replaces 32-bit floating point weights and, potentially, activations with lower bit (*i.e*. lower precision) representations, while aiming to maintain accuracy.

Quantization is often used in neural network *compression*. This aims to reduce the memory occupied by the network weights as much as possible to, for example, lower the overall memory footprint required to store the network. It can also be used to increase neural network inference speed (*fast inference*), when applied to both weights and activations, by substituting expensive floating-point Multiply and Accumulate (MAC) operations with cheaper alternatives such as integer, bitwise operations or addition-only operations.

The best quantization results are achieved when labelled training data is available, as the quantized model can be fitted to the dataset, which feeds the training algorithm with prior knowledge of what the activation maps will look like and what the expected output will be. Maintaining a high accuracy becomes much more difficult when only a pre-trained model is available.

In this paper we focus on this latter scenario, and quantize both weights and activations to produce neural networks that are both smaller and have faster inference. Our key insight is that, in the absence of training data, this can be best achieved by forcing the probability distributions over the weights and activations of the low precision quantized model to mirror those of the original full-precision model. We introduce a novel convolution operator, which we call *DSConv*, that factorises the convolution weights into (i) a low-precision component with the same size as the original kernel and (ii) a high-precision distribution shift component, with a variable size (*e.g*. as small as one *float 32* value per kernel). A similar procedure, inspired by the block floating point approach [35], is used to quantize activations. We also show that accuracy can be improved when using a distillation [19] inspired weight adaptation approach, that uses the original pre-trained model and unlabelled input data.

The main contribution of this paper is the introduction of a convolution operator that (i) serves as a "drop-in and play" replacement for standard convolution and uses low-bit fixed point computation for the bulk of operations without the need of retraining using labelled data, and (ii) provides a hyperparameter that can be tuned to favor accuracy or memory usage/speed of computation for any given task. Our quantization strategy is able to achieve state-of-the-art results, as demonstrated by our experimental section.

The remainder of this paper is structured as follows. §2 presents the previous papers on quantization. §3 explains the method in detail. §4 shows the results of the experiments performed in a variety of architectures and settings. §5 concludes the paper with a discussion on its performance and possible applications and limitations.

2. Related Work

The use of low-bit width networks saves a significant amount of memory and computation, especially when targeted to custom hardware. For example, for 8-bit operations, [26] reports up to 10x increase in speed, and [12] reports up to 30x in energy saving and chip area. We categorize the previous research that tries to increase the neural network efficiency in two groups:

Quantization with labelled data. Most research in neural network quantization has focused on problems that involve retraining, by either starting from scratch or by adapting from an existing pre-trained network. BinaryConnect, BWN and TWN [11, 32, 28] use 1-bit and ternary weights to make the FP-MACS addition only. XNOR-Net and BNN [32, 11] applied 1-bit quantized weight and activations to ImageNet for fast inference, at the cost of a significant drop in accuracy. WRPN [30] improved this accuracy by using wider versions of the same architectures. Early demonstrations of low-bit network acceleration in custom hardware include Ristretto [16], which also uses data to quantize the network to 8-bit models. Many other papers followed, by also training the quantization scheme, using binary basis vectors, such as in LQ-Nets [38], Halfway Gaussian Quantization (HWGQ) [6], and ABC-Net [29]. DoReFa-Net [40] also quantized gradients, alongside weights and activations.

The compression problem has also mostly been dealt with by using retraining with access to labelled data. In DeepCompression [17], two of the three steps of the algorithm require retraining (pruning and quantization), with Huffman Encoding being performed without the need for data. HashedNet [7] use the "hashing trick" to save significant amounts of memory when storing the network, but still require labelled data for tuning. The more recent approach [31], uses distillation [3, 19] obtain compressed weights, but also requires the full labelled training set.

Several approaches introduce novel float data formats. Examples are Dynamic Fixed Point [10], which substitutes the normal floating point numbers with a mix of both fixed and floating point; and Flexpoint [25], which aims to leverage the range of floating point numbers and the computational complexity of fixed point and promises to perform forward and backwards operations with limited range. The idea of substituting the representation of single-precision (FP32) values in favour of other formats is also adopted in the *bfloat16* format in Tensorflow [2], which employs the *binary float 16* format that uses 7-bits for the mantissa instead of the usual 23-bits.

Quantization without labelled data. Whereas the problem of quantizing with labelled data has been researched extensively, the problem of quantizing without data has received far less attention. Recent papers that explore this possibility are [39, 8, 23, 4], which either report results only for 8-bit quantization or employ calibration data of some sort - *i.e*. an unlabelled small fraction of the validation dataset that is used for weight adaptation. Industry approaches have implemented quantization techniques that use only a small amount of unlabelled data, in systems such as TensorRT [1]. In this instance, they can successfully quantize a network to 8-bits with no loss of accuracy (sometimes even with improved accuracy) from 1000s of sampled images [1]. Other examples include the Google TPU, and Project Brainwave ([15, 9]), all of which quantize neural networks to 8-bits for fast inference. Another work that shows that 8-bit quantization does not affect efficiency significantly is [26], where they show that this is true even when quantizing both activation and weights.

In this paper, we show that quantization can be done effectively to 4-bits for both weights and activations, without the need of retraining labelled data, with further potential improvements when using adaptation with unlabelled data.

3. Method

For a given neural network inference $f(x)$, the prediction of $f(Mx)$ should be identical (considering that biases are scaled accordingly), for $M \in \mathbb{R}$. *DSConv* is built on the intuition that this property holds for some nonlinear transforms of x, as long as the relative distribution of the weights and activation values remains the same. We believe one such transform to be quantization, *i.e*. we can scale and bias quantized weights and activations in a way that is friendly for low precision representation and still maintain the same neural network accuracy, as long as distribution over the weights and activations remains unchanged.

Adopting this strategy to the entire 4D tensor would yield a very high cropping error, since a single scaling factor M would not be able to single-handedly capture the entire tensor distribution. In this paper we adopt the more general strategy of using a tensor of scaling factors, whose size is adjusted to capture the range of values with higher fidelity. Every tensor of floating point values is divided into two components: one tensor with the same size of the original, composed of low-bit integer values, and another one with a fraction of the size, composed of floating point scaling factors. Each scaling factor is responsible for the scaling of a subgroup of B integer values along its tensor depth dimension, where B is the block size hyperparameter.

The steps taken by *DSConv* are as follows: **(I)** From a pre-trained network, divide the weight tensor depth-wise into blocks of variable length B and quantize each block; **(II)** Use the block floating point (BFP) format to quantize the activations, where the block is the same size as the weight tensor; **(III)** Multiply the integer values of the activations and the weight tensor to maximize inference speed; **(IV)** Multiply the final values by their respective scales to shift the distribution of the individual blocks to the correct range.

3.1. Weight Quantization

We propose a method for quantizing weights that shares one floating-point value for each block of size B, along the depth dimension of each weight tensor filter. An example for the resulting sizes for each filter can be seen in Figure 1.

Given a weight tensor of size (C_o, C_i, K_h, K_w) and a block size hyperparameter B, we first divide the tensor into two components: the *Variable Quantized Kernel* (VQK), which is composed of low-bit values, and is of the same size as the original tensor; and the *Kernel Distribution Shift* (KDS), composed of single precision numbers ξ, and of size $(C_o, \lceil \frac{C_i}{B} \rceil, K_h, K_w)$, where $\lceil x \rceil$ is the ceiling operation.

The B hyperparameter can seamlessly be modified to accommodate for trade-off between floating point arithmetic and fixed point arithmetic, with $B = 1$ for pure floating point to $B \geq C_i$ for maximum fixed point.

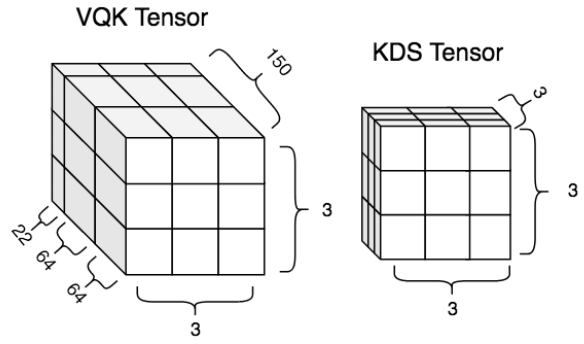


Figure 1. Size of VQK and KDS for each weight filter, for the case of $B = 64$. This reduces the number of FP MACs from 1350 to 27.

The VQK then holds integer values in 2s complement such that for a specific number of bits b chosen, the weights are in the interval:

$$w_q \in \mathbb{Z}, b \in \mathbb{N} \mid -2^{b-1} \leq w_q \leq 2^{b-1} - 1, \qquad (1)$$

This allows all the operations to be performed using 2s complement arithmetic, as explained in §3.3.

By simply changing the normal convolution to *DSConv*, the memory saved per tensor weight is:

$$p = \frac{b}{32} + \frac{\lceil \frac{C_i}{B} \rceil}{C_i} \qquad (2)$$

Equation 2 shows that, for large enough values of B and C_i, the memory saved is approximately the number of bits b divided by 32. For illustration purposes, Table 1 shows the numerical results for realistic values of C_i, B, b and p for some layers of the GoogLeNet [34] architecture. As it can be seen, significant memory saving can be achieved by only quantizing, with no additional method such as Huffman Coding [21].

	Channel (C_i)	Block (B)	Bit (b)	Saving (p)
Inception (4a)	128	64	4	14.1%
Inception (4a)	128	128	4	13.3%
Inception (4a)	128	32	3	12.5%
Inception (4c)	256	128	3	10.2%

Table 1. Memory savings by quantizing only.

Given a known pre-trained model, the weights of each block are stretched and rounded to fit in the interval in Equation 1, and they are stored in the VQK. Next, we explored two possible methods to calculate the KDS values: (i) minimizing the KL-Divergence, which seeks to find the minimum loss of information between the distribution of the original weights and the kernel distribution shifter and emphasizes the idea that the resulting VQK, after being shifted, should have a similar distribution to the original weights; or (ii) minimizing the L2 norm (Euclidean distance), which can have the interpretation that parameters should be the closest to the optimum value of the original network.

To minimise the KL-Divergence we first take the softmax values of both the shifted VQK and the original distributions:

$$T_j = \frac{e^{w_j}}{\sum_i e^{w_i}}, \; I_j = \frac{e^{\hat{\xi} \cdot w_{q_j}}}{\sum_i e^{\hat{\xi} \cdot w_{q_i}}} \qquad (3)$$

We then use gradient descent to minimize the following for each slice:

$$\xi = \min_{\hat{\xi}} \sum_j T_j \log \left(\frac{T_j}{I_j} \right), \quad \forall \, (1, B, 1, 1) \text{ slices} \qquad (4)$$

where ξ is the KDS value for that block.

The other method minimises the following L2 norm for each slice:

$$\xi = \min_{\hat{\xi}} \sum_{i=0}^{B-1} (w_{q_i} \, \hat{\xi} - w_i)^2 \qquad (5)$$

which has the closed form solution:

$$\therefore \; \xi = \frac{\sum_{i=0}^{B-1} w_i w_{q_i}}{\sum_{i=0}^{B-1} w_{q_i}^2}, \qquad \forall \, (1, B, 1, 1) \text{ slices} \qquad (6)$$

In practice, both strategies produced approximately equal values. We performed all the experiments using the L2 norm approach, since it has a closed form solution.

Algorithm 1 summarizes the process of initializing both the VQK and the KDS given a pre-trained network model.

3.2. Activation Quantization

Our approach aims to achieve good performance in the absence of any training data. This means that we have **no prior knowledge** of what values or distribution the activation maps will have. Therefore, this quantization cannot be data-driven. Instead, we used an approach inspired by the block floating point (BFP) method of [35, 33, 14, 9, 15].

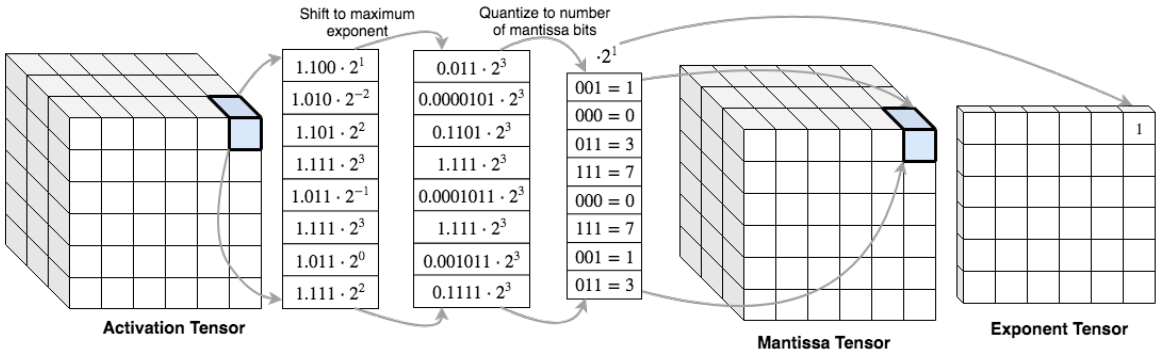


Figure 2. Example of quantizing activation. This is the specific case where the mantissa bit was set to 3 and the block hyperparameter was set to 8. Note that ½ LSB rounding performed when cropping. Note that this is performed after the ReLU layer, which means that all values are unsigned positive.

Algorithm 1 Weight Initialization

Input bit-length b, pre-trained weights $\mathbf{w}$, block size B

procedure QUANTIZE
  $m \leftarrow 2^{b-1} - 1$
  for all Block B **do**:
    $w_m \leftarrow \arg\max_{\mathbf{w}}(|w|)$
    $s \leftarrow m/w_m$
    for all w_i in B **do**:
      $w_q \leftarrow \text{round}(w_i \cdot s)$
    $\xi = \sum_{i=0}^{B-1} w_i w_{q_i} / \sum_{i=0}^{B-1} w_{q_i}^2$
  return ξ, $\mathbf{w_q}$

Figure 2 shows our activation quantization approach. For a given mantissa width, the *activation tensor* is divided into blocks, and, for each block, we find the maximum exponent. The mantissa values of all the other activations in the block is shifted such that they match the maximum exponent, which is then cropped (using ½ LSB rounding) to match the number of specified bits. This results in two tensors: a *mantissa tensor*, which has the same shape as the original tensor, but populated with b bits; and an *exponent tensor*, which has size $(C_o, \lceil \frac{C_i}{B} \rceil, H, W)$.

We call this a BFP approach because we are essentially "sharing" the exponent for each block of size B. This allows for a control over how coarse the quantization is, and how much cropping error we are willing to accept to get the lowest bit-length for the mantissa tensor.

This approach has the added benefit of allowing low-bit integer operations between the weights and activations, as we show in §3.3. Therefore, the trade-off between efficiency and speed of computation is as follows: the higher the value of B, the bigger the cropping error will be, but the exponent tensor and the KDS will be shallower. This is a different trade-off to the number of bits b, which adds more computational complexity and memory to the mantissa tensor. The values of b and B are then inversely proportional to each other and counter-balance each other's positive and negative effects. The goal then becomes to get the most accuracy with the lowest number of mantissa bits b and largest value for the block size B.

3.3. Inference

During inference, the hardware can take advantage of the fact that the VQK and the mantissa tensors are low-bit integer values, which allows it to save time performing integer operations rather than floating point operations. The data path is illustrated in Figure 3.

First, each of the blocks of the VQK and the mantissa tensor are dot producted, resulting in one value each. All of these operations can be conducted in low-bit fixed point arithmetic, which saves significant processing time. At the end of the block multiplications, the result is a tensor of the same size as both the exponent tensor and the KDS.

The exponent tensor is merged with the KDS tensor by adding its value to the exponent of the KDS tensor values. This results in a tensor of the same size of floating point numbers. Finally, this tensor multiplies the result of the product of the VQK and the KDS, and yields a single floating point number as the output activation.

Notice that the inference is as highly parallelizable as a standard convolution, but instead of performing most of the multiplications using floating point arithmetic, the majority can be substituted by integer multiplications, saving energy, bandwidth and computation time.

This also means that, for each weight and activation multiplication, the number of blocks is proportional to the number of total floating point MAC operations, and the size of the tensor itself gives the number of INT MAC operations.

Batch Normalization Folding. Similar to [23], we perform "folding" of the Batch Normalization (BN) [22] parameters in models that have them. Since batch normalization has been shown to improve training (see [22]), we keep it during the training phase and only fold it for inference.

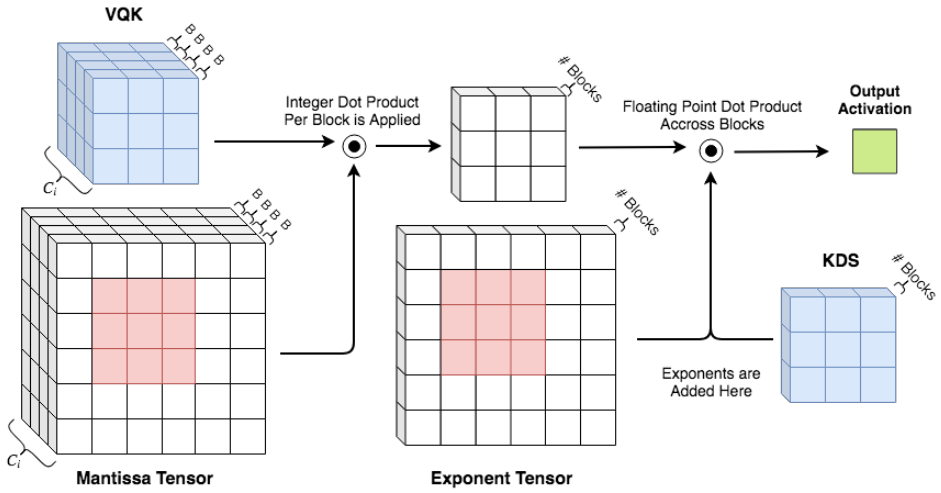


Figure 3. Example of convolution being performed, with VQK tensor (in blue) multiplying one section of the Mantissa Tensor (in red). Each block of the VQK performs a dot product with each block of the Mantissa tensor. The result is a tensor with depth equal to the number of blocks depth-wise. The Exponent Tensor performs addition of the exponent value of the KDS, and the result is multiplied by the result of the dot product of the VQK and the Mantissa tensor, and that becomes the final output activation. This is performed for every filter.

When folding the BN parameters, we do so with the KDS, since they are unique per channel and use FP32 values. We perform the folding using the equations:

$$\xi_{fold} = \frac{\xi\gamma}{\sqrt{\sigma_B^2 + \epsilon}} \tag{7}$$

$$b_{fold} = \beta - \frac{\gamma\mu_b}{\sqrt{\sigma_B^2 + \epsilon}} \tag{8}$$

where the parameters γ, σ, ϵ, β and μ are as defined in [22], ξ is the KDS tensor and b_{fold} is the resulting bias of the *DSConv*.

3.4. Distillation for Unlabelled Data Adaptation

It is often the case that unlabelled data may be available, as shown by the vast array of unsupervised learning methods available. For this specific scenario, we adopt a strategy similar to [19]. We use the distillation loss without labelled data to try to regress the FP32 model to the quantized one, by using the FP32 logits as the target, and minimizing the loss for regression.

We create a "shadow" model which holds single-precision numbers. Before each inference, this model is quantized to the VQK and KDS, inference is performed and the gradients are calculated. During the update phase, the gradients are accumulated as single-precision numbers, and the method is performed until convergence.

Quantizing the activation maps after each inference would cause the gradients to be zero everywhere. To avoid this problem, we use the Straight-Through Estimator (STE) [5, 37], to calculate the backwards gradient. Particularly, we use the ReLU STE since it was shown in [37] that it gives a better accuracy than using the Identity STE for deeper networks. The gradient is then also accumulated in a "shadow" FP32 model, which is quantized after each batch iteration.

We use the ADAM Optimizer [24], with initial learning rate 10^{-5} and after the loss plateaus, this rate is changed to 10^{-6}. All other hyperparameters and data augmentation details follow their respective original papers.

We use 960 images (30 batches of 32) from the validation dataset for the distillation, and we test the accuracy using the rest of the images (49,040 images in total).

4. Experiments and Results

We tested our method on various neural network architectures: ResNet [18], AlexNet [27], GoogleNet [34], and DenseNet [20]. We benchmarked our results on the ImageNet dataset [13] (more specifically ILSVRC2012), which has 1.2M images in the training set and 50k images in the validation set. The results reported use images drawn from the validation set. We tested our algorithm for all the tasks indicated in the introduction. This section continues as follows: §4.1 finds the theoretical computational saving for DSConv; §4.2 shows the results without training or adaptation; §4.3 shows the accuracies when the model is adapted with unlabelled data; and §4.4, for comparison with previous methods, shows the results for the retraining performed in DSConv using labelled data.

4.1. Theoretical Computational Load on Block Size

Computational load is traditionally reported as a function of number of MAC operations needed in order to com-

plete the algorithm. We note two caveats: integer MACs are far less complex than FP MACs and, when supported by a hardware implementation, can be run orders of magnitudes faster than FP operations [26]; our method also relies on the ability to create the mantissa tensor and the exponent tensor dynamically (the VQK and the KDS are created statically, so they are not considered here). This requires `MAX`, `SHIFT` and `MASK` operations. These can be implemented efficiently in custom hardware with few clock cycles. Therefore we will focus on the comparison between number of INT vs FP operations to assess the advantage of using this method.

In order for our method to be faster than normal convolution, the time spent to perform the INT operations must be less than the time spent on the FP32 operations. This difference is a function of the block size and on the channel parameter. Equation 9 shows the relation between the time for an INT operation and the time for an FP operation:

$$T_{int} \leq T_{FP} \frac{C_i - \lceil \frac{C_i}{B} \rceil}{C_i(1+\eta)} \tag{9}$$

The values T_{int} and T_{FP} capture the amount of time needed to perform an INT and an FP operation, respectively. The parameter η is an "ideality" parameter that represents the overall overhead in the `MAX`, `SHIFT` and `MASK` operations to perform *DSConv* in comparison to the normal convolution operator.

Also notice that, if C_i is divisible by B (which is often the case), then Equation 9 becomes independent of the channel size and simplifies to:

$$T_{int} \leq T_{FP} \frac{1 - \frac{1}{B}}{1+\eta}, \text{ if } B \mid C_i \tag{10}$$

Table 2 shows the ratio $1 - \frac{1}{B}$ for the most common block sizes experimented, when $\eta = 0$. As can be seen, if the time to compute an INT value is less than 0.75 of the time to compute a floating point operation, then all block sizes bigger than 4 will be faster than the normal convolution. This is often likely to be the case. For example, in modern CPUs and in some GPUs, 8-bit operations can be up to 10x faster than FP32 operations [26], and lower bit operations can potentially be even faster in custom hardware such as FPGAs. In custom software, operations in less than 8-bit are also often faster.

Block	4	8	16	32	64	128
Ratio	0.750	0.875	0.938	0.969	0.984	0.992

Table 2. Relationship of Block Size and speed ratio needed.

The block size B imposes a limit on how much faster *DSConv* can be over traditional convolution operators. Naturally, *DSConv* can be up to $min(C_i, B)$ times faster than the traditional convolution, since it has $min(C_i, B)$ times less floating point operations than a normal convolution. For example, for block sizes of 128 to 256 and channel sizes of more than 256, *DSConv* can be up to two orders of magnitude faster than a normal convolution.

4.2. Accuracy Before Retraining or Adaptation

Our method is designed to produce accurate results even when training data is not available, by quantizing from a pre-trained network.

		Accuracy (% @ Top1 and Top5)					
Block	**Bit (W/A)**	**ResNet50**		**ResNet34**		**ResNet18**	
-	32 / 32	76.1	92.9	73.3	91.4	69.8	89.0
256	8 / 32	76.1	92.9	73.3	91.4	69.7	89.0
128	6 / 32	75.9	92.8	73.2	91.4	69.5	89.0
16	4 / 32	75.1	92.3	72.6	91.0	67.7	87.8
4	2 / 32	65.1	86.2	66.8	87.6	59.1	81.7
128	8 / 8	76.1	92.9	73.3	91.4	69.7	89.1
64	6 / 6	75.9	92.8	73.2	91.4	69.6	89.0
64	5 / 5	75.4	92.6	72.7	91.0	68.9	88.5
16	4 / 4	74.8	92.1	72.3	90.8	67.3	87.7

		Accuracy (% @ Top1 and Top5)					
Block	**Bit (W/A)**	**GoogLeNet**		**VGG19**		**Dense121**	
-	32 / 32	67.6	88.3	72.4	90.9	74.4	92.0
256	8 / 32	67.6	88.3	72.3	90.9	74.4	92.0
128	6 / 32	67.1	88.0	72.4	90.9	74.2	91.8
16	4 / 32	63.3	85.4	72.1	90.7	72.9	91.2
4	2 / 32	27.6	51.1	69.4	89.0	62.7	84.4
128	8 / 8	67.6	88.3	72.3	90.9	74.4	92.0
128	6 / 6	67.1	88.0	72.4	90.8	74.2	91.8
64	5 / 5	65.5	86.8	72.3	90.8	73.8	91.6
16	4 / 8	63.3	85.4	72.1	90.7	72.9	91.2

Table 3. Accuracy of Fast Inference and Compression without data as a function of Bit width (in Weights and Activation) and Block Size.

The second and fifth rows of Table 3 show that for both the compression and fast inference problems, no loss of accuracy can be achieved with 8-bit networks even with very high block sizes, as already demonstrated by previous papers and real-life applications [26, 16]. The results also shows that compression down to 4-bits (which in convolutions with channel size input of 256 would yield a **5x** compression rate) results in an accuracy drop of only 1% to 2% depending on the architecture. It can also be seen that very low-bit quantizations become noticeably unstable, varying greatly with architecture. At the extreme, using 2-bits, losses vary by as much as -40% for GoogLeNet and only -11% for ResNet50.

The last four rows show the results for the fast inference problem. Also as known in previous research papers [26, 16], models of 8/8 bits lose only around 0.1% accuracy. For models of 5/5 and 4/4, we get a drop of 1% to 3% in accuracy. To our knowledge, this is the smallest bit-

width for fast inference that has been reported when models are neither retrained nor adapted.

The variance with respect to architecture suggests that quantization for 5 or less bits is unstable. However, even for fast-inference with 8-bit accuracy, it can achieve stable and satisfactory results within 1% of the full precision model.

Accuracy with respect to Block Size Table 4 shows the accuracy with respect to block size. The table shows the results of quantizing the weights only, where the number in parenthesis represents the bit-width of the weights. Naturally, this represents a trade-off between memory and computational load against precision of the network. The largest discrepancy in accuracy can be seen in models that use 3 or 2 bit weights. For example, the GoogLeNet model with 3-bits improves its Top1 accuracy from 5.7% to 56.8% when changing from a block-size of 256 to 8.

Block	256	128	64	32	16	8	4
ResNet50 (4)	73.0	73.5	73.8	74.7	75.1	75.4	75.6
ResNet50 (3)	44.6	51.9	59.6	67.4	69.6	73.6	74.7
ResNet34 (4)	70.8	70.8	71.5	71.9	72.6	72.8	72.9
ResNet34 (3)	59.5	60.4	63.6	66.8	69.2	70.6	71.6
GoogLeNet (4)	52.5	57.0	59.1	61.7	63.3	65.6	66.5
GoogLeNet (3)	5.7	22.4	37.6	40.3	49.2	56.8	62.5
VGG19 (3)	67.6	68.6	69.5	70.4	71.1	71.6	71.8
VGG19 (2)	11.3	21.8	38.1	55.5	63.1	67.5	69.4

Table 4. Accuracy with respect to block size for the compression case with no data available.

When using 4-bit quantization schemes, a decrease in the block size achieves accuracy levels that are within 1% to 2% of the full precision network. This is the case for example for most networks with block sizes of 16 to 32.

4.3. Accuracy Adapted using Unlabelled Data

The results when adapting our network with extra unlabelled data are reported in Table 5. For each Block-Bit configuration, two results are reported: on the top we show the result before adaptation and the bottom (in bold) the result after adaptation using unlabelled data. This strategy increases inference accuracy using 4-bits only for the weights and for the activations to within 2% of the FP32 precision of the network, even for the extreme cases of using 128 as the block size.

For 3-bits, even though we recover up to 30% accuracy, there is still a considerable gap between the low-bit accuracies and the full precision ones. For ResNet50 this gap is of 6% whereas for GoogLeNet it can reach 10%.

Table 6 shows results of recent papers (introduced in the literature review) that use calibration. This is similar in spirit to our adaptation stage, in that both approaches use only unlabelled data. The notable exception is that we use distillation to convert a full-precision model to a low-precision model, whereas the other approaches generally calibrate just the optimal clipping strategy. It can be seen that, even using a big block size of 64, we achieve better performance. To our knowledge, this is the best result achieved for fast inference using only adaptation data.

		Accuracy (% @ Top1 and Top5)							
Block	**Bit (W/A)**	**ResNet50**		**ResNet34**		**ResNet18**		**GoogLeNet**	
32	4 / 4	74.1	91.8	71.3	90.2	66.4	87.1	61.2	81.9
		74.8	**92.1**	**71.8**	**90.6**	**68.3**	**88.1**	**66.1**	**87.2**
64	4 / 4	73.0	89.8	70.9	88.4	66.1	85.1	58.4	79.3
		74.8	**92.1**	**71.8**	**90.6**	**68.4**	**88.1**	**65.2**	**86.8**
128	4 / 4	72.6	89.6	70.2	87.9	65.8	84.8	55.8	77.3
		74.2	**92.0**	**71.3**	**90.5**	**67.5**	**87.8**	**64.7**	**86.3**
32	3 / 3	63.3	85.0	63.2	85.0	55.3	78.4	34.5	60.5
		72.6	**91.1**	**69.6**	**89.4**	**66.8**	**87.5**	**60.0**	**83.3**
64	3 / 3	54.4	77.9	58.1	81.3	30.1	51.6	29.3	53.9
		71.5	**90.4**	**69.1**	**89.3**	**65.8**	**87.0**	**56.7**	**81.0**

Table 5. Results of adaptation for a variety of architectures for the case where an adaptation dataset is provided.

			VGG16	AlexNet	ResNet18	ResNet50
	W	A	Top1	Top1	Top1	Top1
Naive [4]	4	8	29.0%	1.8%	0.8%	0.4%
CW [4, 26]	4	8	70.2%	52.9%	59.3%	72.4%
K+B [4]	4	8	70.0%	54.7%	67.0%	74.2%
OCS+MSE [39]	5	8	-	-	-	73.4%
Ours NA (16)	4	8	**71.3%**	**55.9%**	**67.6%**	**75.1%**
Ours NA (32)	4	8	**71.2%**	**55.4%**	**66.7%**	**74.7%**
Naive [4]	8	4	53.9%	41.6%	53.2%	52.7%
KLD [4, 1]	8	4	67.0%	49.6%	65.1%	70.8%
ACIQ [4]	8	4	70.5%	55.2%	68.9%	74.8%
Ours NA (16)	8	4	**71.5%**	**56.4%**	**69.6%**	**75.7%**
Ours NA (32)	8	4	**71.5%**	**56.4%**	**69.6%**	**75.6%**
Naive [4]	4	4	23.7%	1.8%	0.6%	0.4%
ACIQ [4]	4	4	68.9%	53.0%	65.3%	72.6%
OMSE+O [8]	4	4	-	54.5%	67.4%	72.6%
Our (64)	4	4	**71.1%**	**55.8%**	**68.4%**	**74.8%**

Table 6. Adaptation of our method vs previous papers. The Naive method refers to simple clipping. CW is Channel-Wise quantization adopted in [36, 26]. K+B is the K-Means + Bias method of [4]. KLD is the KL-Divergence method first proposed in [1]. OMSE+O is the OMSE + offset method of [8]. "Ours NA" refer to our method with No Adaptation

4.4. Accuracy After Labelled Data Retraining

We also compared *DSConv* with previous methods that retrain/finetune with labelled data (the vast majority in the literature). Training happens similarly to adaptation, but now we use labelled data and use cross-entropy loss in the classification error instead of using logits.

Table 7 shows the results for ImageNet on a variety of architectures. As many previous papers report different initial FP accuracy for the same architecture, we have also in-

ResNet 18

	W	A	Top1	Top5
FP	32	32	69.6	89.2
FP_{LQ} [38]	32	32	70.3	89.5
BWN [32]	1	32	60.8	83.0
TWN [28]	2	32	61.8	84.2
TWN [28]	2	32	65.3	86.2
TTQ [41]	2	32	66.6	87.2
LQ [38]	2	32	68.0	88.0
Ours (32)	2	32	**68.7**	**86.7**
LQ [38]	3	32	69.3	88.8
Ours (32)	3	32	**69.7**	**87.5**
LQ [38]	4	32	70.0	89.1
Ours (32)	4	32	**70.0**	**87.6**
XNOR [32]	1	1	51.2	73.2
DoReFa [40]	1	2	53.4	-
DoReFa [40]	1	4	59.2	-
Ours (32)	1	4	**65.2**	**86.2**
HWGQ [6]	1	2	59.6	82.2
ABC [29]	3	3	61.0	83.2
ABC [29]	5	5	65.0	85.9
Ours (128)	5	5	**70.0**	**89.3**
LQ [38]	1	2	62.6	84.3
LQ[38]	2	2	64.9	68.2
LQ [38]	3	3	68.2	87.9
Ours (16)	3	3	**69.2**	**88.9**
LQ [38]	4	4	69.3	88.8
Ours (64)	4	4	**69.8**	**89.2**

DenseNet121

	W	A	Top1	Top5
FP	32	32	75.0	92.3
DoReFa [40]	2	2	67.7	88.4
FP_{LQ} [38]	32	32	75.3	92.5
LQ [38]	2	2	69.6	89.1
FP_{Ours}	32	32	74.4	92.2
Ours (32)	2	32	**74.0**	**91.8**
Ours (16)	2	2	**72.1**	**90.6**

AlexNet

	W	A	Top1	Top5
FP	32	32	57.1	80.2
TWN [28]	2	32	54.5	76.8
FP_{LQ} [38]	32	32	61.8	83.5
LQ[38]	2	32	60.5	82.7
FP_{Ours}	32	32	**56.6**	**79.1**
Ours (32)	2	32	**55.0**	**78.1**

ResNet34

	W	A	Top1	Top5
FP	32	32	73.3	91.3
FP_{LQ} [38]	32	32	73.8	91.4
Ours (32)	3	32	**73.4**	**90.1**
Ours (32)	4	32	**73.6**	**90.1**
HWGQ [6]	1	2	64.3	85.7
Ours (64)	1	4	**68.2**	**86.8**
ABC [29]	3	3	66.7	87.4
ABC [29]	5	5	68.4	88.2
Ours(16)	4	4	**73.0**	**89.7**
LQ[38]	1	2	66.6	86.9
LQ [38]	2	2	67.8	89.1
LQ [38]	3	3	71.9	90.2
Ours (16)	3	3	**72.7**	**89.6**

ResNet50

	W	A	Top1	Top5
FP	32	32	76.0	93.0
FP_{LQ} [38]	32	32	76.4	93.2
LQ[38]	2	32	75.1	92.3
Ours (32)	2	32	**75.2**	**92.6**
LQ[38]	4	32	76.4	93.1
Ours(128)	4	32	**76.4**	**93.0**
HWGQ [6]	1	2	64.6	85.9
ABC [29]	5	5	70.1	89.7
LQ[38]	1	2	68.7	88.4
LQ[38]	2	2	71.5	90.3
Ours(32)	2	2	**72.5**	**91.2**
LQ[38]	3	3	74.2	91.6
Ours (32)	3	3	**75.2**	**92.4**
LQ[38]	4	4	75.1	92.4
Ours (64)	4	4	**76.2**	**92.9**
Ours (128)	4	4	**76.1**	**92.8**

GoogLeNet

	W	A	Top1	Top5
FP_{HWGQ} [6]	32	32	71.4	90.5
HWGQ [6]	1	2	63.0	84.9
FP_{LQ} [38]	32	32	72.9	91.3
LQ [38]	1	2	65.6	86.4
LQ [38]	2	2	68.2	88.1
FP_{Ours}	32	32	67.6	86.3
Ours (32)	4	4	**66.3**	**85.5**
Ours (64)	4	4	**65.7**	**85.1**

Table 7. Results of retraining for a variety of architectures. Table derived from [38]

cluded the initial FP of the single precision results to make an evaluation that takes into account the "upper limit" of the architecture itself.

From the results, it can be seen that our method can beat the state-of-the-art for a variety of cases, as long as the Block Size is adjusted properly to give more emphasis on accuracy rather than speed.

DSConv can beat the state-of-the-art when using bit sizes that are either 4 or 5. In these cases (such as ResNet18 using 5/5, ResNet50 using 4/4 and GoogLeNet using 4/4), we also use a large Block Size, with slightly better than the FP efficiency in ResNet18 when using $B = 128$ and bit sizes of 5/5, and $B = 64$ for bit sizes of 4/4.

In order to get state-of-the-art results for 3-bits or less, a lower block size is needed. This is shown for DenseNet121 results, which uses bit-width of 2 and Block Size of 16 to get 72.1% accuracy. Extremely low-bit weights and activations do not work very well because the assumption that lower information loss in quantization corresponds to higher accuracy starts to break down. This is supported by the fact that the state-of-the-art approaches for 1 and 2 bit weights are trained from scratch, which suggests that for these cases, quantizing from a pre-trained network is not ideal.

We also show good results for the compression case. ResNet50 with 4 bit and $B = 128$ illustrates that no loss of accuracy is observed, and even using only 2 bits, with accuracy staying within 1% using $B = 32$.

5. Conclusion

We presented *DSConv*, which proposes an alternative convolution operator that can achieve state-of-the-art results whilst quantizing models to up to 4-bits in weight and activation without retraining or adaptation.

We showed that our method can achieve state-of-the-art results without retraining in less than 8 bit settings, which makes it possible for fast inference and less power consumption for rapid deployment in custom hardware. By having the advantage of being tunable by the block size hyperparameter and not needing any training data in order to run, we propose that this method is very suitable for acceleration of convolutional neural networks of any kind.

When using unlabelled data and distillation from the FP32 model, we can achieve less than 1% loss using 4-bit for both weights and activations. Also, as in previous methods, we demonstrate that the assumption that lower information loss in weight quantization correspond to higher inference accuracy breaks down when quantizing to extremely low bits (1, 2 or 3 bits). In these cases, retraining seems inevitable since they are quintessentially different than higher accuracy models.

6. Acknowledgements

This research was partially supported by Intel (previously Omnitek), and we thank our colleagues from the Programmable Solutions Group who greatly assisted in this work. Victor Adrian Prisacariu would like to thank the European Commission Project Multiple-actOrs Virtual Empathic CARegiver for the Elder (MoveCare).

References

[1] NVidia TensorRT. http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf. Accessed: 2019-09-06.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2654–2662. Curran Associates, Inc., 2014.

[4] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment, 2018.

[5] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[6] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[7] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[8] Yoni Choukroun, Eli Kravchik, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference, 2019.

[9] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, Mar 2018.

[10] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

[12] William Dally. High-performance hardware for machine learning, 2015.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. 2009.

[14] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point, 2018.

[15] 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 *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[16] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems*, (99):1–6, 2018.

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

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[19] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[20] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[21] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[25] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*, pages 1742–1752, 2017.

[26] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

[29] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.

[30] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. Wrpn: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134*, 2017.

[31] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[32] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *Lecture Notes in Computer Science*, page 525–542, 2016.

[33] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[35] James H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Inc., New York, NY, USA, 1994.

[36] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks, 2018.

[37] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley J. Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. In *International Conference on Learning Representations*, 2019.

[38] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *Lecture Notes in Computer Science*, page 373–390, 2018.

[39] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting, 2019.

[40] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[41] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization, 2016.

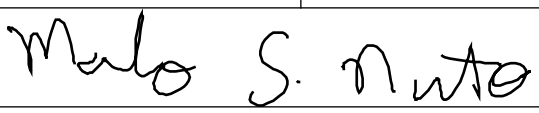
Statement of Authorship for joint/multi-authored papers for PGR thesis

To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	DSConv: Efficient Convolution Operator
Publication Status	X Published □ Accepted for Publication □ Submitted for Publication □ Unpublished and unsubmitted work written in a manuscript style
Publication Details	Nascimento, Marcelo Gennari do, Roger Fawcett, and Victor Adrian Prisacariu. "DSConv: efficient convolution operator." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019.

Student Confirmation

Student Name:	Marcelo Gennari do Nascimento		
Contribution to the Paper	• Contributed in the conception of the idea together with both co-authors • Designed, and wrote the code for all experiments • Performed all the experiments • Created all of the tables and diagrams • Wrote the manuscript • Presented the poster at the Conference		
		Date	12/09/2022

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Victor Adrian Prisacariu			
Supervisor comments The description on contributions is accurate			
Signature		Date	12/09/2022

This completed form should be included in the thesis, at the end of the relevant chapter.

3.5 Conclusion

In this chapter, we introduced *DSConv: Efficient Convolution Operator* as a method to quantise the convolutional layers of a neural network. It works by using the Block Floating Point format, where one exponent is shared channel-wise across a range of mantissa values. This allows the network to both have a wide range of representation for its numbers (due to the exponent) and also be able to perform the bulk of operations in integer format (due to the mantissas).

We tested our algorithm in many different settings: we first evaluated its usefulness as a Post Training Quantization method, where no retraining is necessary, and we showed that by using blocks of 64 we can compress both weights and acivations of a ResNet50 down to 6 bits and lose only 0.2% accuracy on ImageNet. We also evaluated our method using unlabelled data (a subset of 960 images was used), and we were able to bring the bitwidth down to 4 for a ResNet50 while losing less than 2% accuracy on ImageNet. We also tested our method using Quantized Aware Training and were able to have no loss of accuracy even with blocks of 128 when quantizing ResNet50 on ImageNet.

The main shortcoming of this method is that it can only be efficiently implemented if appropriate hardware is designed to run this algorithm. A potential avenue to expand on this work is to find ways of quantising different layers of the network to different precision, which is the problem tackled in the next chapter.

4

Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes

Contents

4.1 Context 42
4.2 NAS Algorithms 42
4.3 Author Contribution 46
4.4 Publication: Gennari do Nascimento *et. al.* "*Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes*" ECCV 2020 46
4.5 Conclusion 64

In this chapter, we will mainly present our second paper "Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes", published at the European Conference of Computer Vision, in 2020. We will first start by giving some context on the motivation of our work, and provide some background information on the different types of NAS algorithms in order to justify our choice of Bayesian Optimization for our specific problem. We will then present the paper, and conclude with a summary of our contributions, and what has been achieved.

4.1 Context

Previous papers, including our work *DSConv* presented in Chapter 3, have mostly assumed that all layers of a CNN should be quantized to the same bitwidth. From a hardware perspective, this indeed makes sense as the design is greatly simplified under that assumption. However, there is evidence that different layers of a neural network are responsible for different levels of abstraction [110], which indicates that they can be compressed at different ratios. Therefore, quantising all layers to the same bitwidth means that some layers are overquantised, while other layers are underquantised.

The problem with quantising CNNs to different bitwidths lies in choosing the precision with which each layer should be quantised. Considering a CNN with n layers, where each can be quantised anywhere from 1 to 8 bits, the total number of combinations to try is $8^n = 2^{3n}$, which grows exponentially with the number of layers. Brute-force / random search therefore is not an option.

4.2 NAS Algorithms

There are three often used methods that can optimize a black-box function: Reinforcement Learning (RL), Evolutionary Algorithms (EA), and Bayesian Optimization (BO). This problem setting is very similar to NAS, which has a long history, as discussed in the literature review in Chapter 2. We are going to briefly give an overview of each of these NAS methods to justify our choice of using BO for our specific application.

Reinforcement Learning for NAS

Reinforcement Learning can be seen as a search problem. Making an analogy with Black Box Optimization, we can think of the environment as a Black Box Function $s_{t+1}, r = f(a|s_t)$ and the goal is to find the values of a that maximizes the expected reward, where s is the state of the environment, a is the action to be taken, and

r is the reward. Since not all functions are differentiable, the RL algorithm can help us find maxima of problems that are not trivial.

This is where Reinforcement Learning has proven to be the most useful: in problems that usually require sequential decision-making, where a model of the environment can be computationally simulated, not trivially differentiable, and where the number of possible states is large [111]. That is why most notable successes for Reinforcement Learning are games that are easy to simulate but hard to play. This includes Backgammon [112], Go [113], Chess [114], and even Star Craft II [115]. For that, we need to only provide the environment, which gives the state, a reward function, and therefore a search space.

This can be very useful for NAS, as it fulfills most of the requirements where Reinforcement Learning can be of use. It can have large finite search spaces (for example, by considering all of the different combinations of kernel size, strides, padding, etc); an environment that can be sampled is available - which removes the need to know the transition probabilities *a priori*; it is model-free, and therefore it is not trivially easy to find an optimum solution, since we have experienced in the past networks that are deeper than others achieving lower accuracy, for example.

Many papers have used RL for NAS. One of the first ones to achieve state-of-the-art results is [37], which uses an RNN and a Policy Gradient method based on the REINFORCE algorithm [116]. As expected, this paper required an absurd amount of computation, having used 800 GPUs for 28 days. An updated version of that is [117], which restrained the search space and used the PPO algorithm instead [118]. They could achieve better results with a less absurd amount of computation, having used 500 GPUs for 4 days.

Since then, many other methods have restructured the search space in order to make the search algorithm faster. A notable example is MNasNet [45], which proposes a platform-aware search space. Other methods include [40] and [119], both of which require significantly less GPU-hours to find good results. To the best of our knowledge, there is one RL paper aimed at finding optimal bit-width per layer of a CNN, which is ReLeQ [120]. This method also uses a Policy Gradient

approach, however it is tested only with shallow networks mostly in the CIFAR10 dataset. We believe that more work can be done in this regard.

Evolutionary Algorithms for NAS

Evolutionary algorithms are a class of search methods that take inspiration from Darwinian natural selection [121]. The intuition is to first have an initial pool of candidates, and by using crossover, mutation, and fitness evaluation, select a new pool of candidates that perform well under certain constraints [121]. The algorithm then iterates through this process until a suitable candidate (or pool of candidates) is found to be optimal. In this context, evolutionary algorithms have been proposed to explore the topology of a network long before the explosion in modern NAS research [122–124].

An interesting property of EAs is that it is naturally a population-based algorithm: it always keeps a pool of potential networks, which can be completely different from each other [125, 126]. This allows the designer to easily explore locally along the Pareto Front of a multi-task problem. This means that instead of providing one solution, EAs can offer a set of solutions that are along this Pareto Front [127].

Another interesting property of EAs is how easy it is to sample new neural networks. Often, NAS methods require either a predefined super-graph, or a limited search space, for example by restricting the search to cell-based methods [125]. EAs allow for global search over an unrestricted search space, which can yield architectures that would be outside of the search space of most other methods.

One major drawback of EAs is that they require retraining every time a mutation is performed. There are multiple ways that researchers have tried to solve this problem, from inheriting weights from parent nodes [128], or using a predictor of accuracy [129], for example by very early-stopping, and halting the training of the neural network after only a couple of epochs before evaluation in the validation dataset. However all of these methods can considerably hinder the estimation of the models fitness, since they are only estimates of the real network performance.

Bayesian Optimization for Hyperparameter Search

Another alternative solution is Bayesian Optimization, which uses a probabilistic approach to the search algorithm, by usually assuming a Gaussian Process prior [130]. BO arguably fits the NAS problem better since it is designed to optimize block box functions with very expensive evaluation. It consumes more computation per datapoint, but it pays off since less observations are generally needed in order to get good predictions.

To the best of our knowledge, BO methods have mainly being applied to the closely related problem of hyperparameter search. Most notably, BO has been clearly demonstrated to work in [131], where the authors have tuned 9 hyperparameters of a 3-layer CNN in the CIFAR-10 dataset. Other examples include [132–134].

The BO algorithm falls short in high-dimensional space and when a large number of datapoints is provided. This is because in order to work, the BO algorithm needs to invert a kernel matrix that grows quadratically with each datapoint. Many algorithms have been proposed in order to circumvent this problem [135–137], but the difficulty in implementation and the shortcomings mentioned above have likely hindered researchers from exploring this path for NAS. To the best of our knowledge, there is one previous work on applying BO to NAS, which is NASBOT [138]. This method proposes a new distance metric, which can estimate the "distance" between two neural network architectures in order to evaluate the kernel.

The problem with RL and EA is that they require a lot of feedback from the system in order to work well: they are relatively *sample-inefficient* [139, 140]. Their success in tasks such as playing games is due to the constant and quick feedback given by the environment. For our purpose, we want to maximize the utility of each signal, since training a network is expensive but necessary for every time we sample from the search space. In this set of constraints, Bayesian Optimization is the best framework, which we used to propose the paper presented next.

4.3 Author Contribution

- We propose a method for constraining the search of bitwidth through a CNN by parameterizing it as Bezier and Chebyshev functions
- We cast ths problem as a Hyperparameter Search and apply a sample efficient bayesian optimization method with a Multi-Fidelity Gaussian Process prior
- We thoroughly experiment on large datasets such as ImageNet and using large architecutres such as ResNet50

4.4 Publication: Gennari do Nascimento *et. al.* *"Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes"* ECCV 2020

Check for updates

Finding Non-uniform Quantization Schemes Using Multi-task Gaussian Processes

Marcelo Gennari do Nascimento, Theo W. Costain( ), and Victor Adrian Prisacariu

Active Vision Lab, University of Oxford, Oxford, UK
{marcelo,costain,victor}@robots.ox.ac.uk
https://code.active.vision

Abstract. We propose a novel method for neural network quantization that casts the neural architecture search problem as one of hyperparameter search to find non-uniform bit distributions throughout the layers of a CNN. We perform the search assuming a Multi-Task Gaussian Processes prior, which splits the problem to multiple tasks, each corresponding to different number of training epochs, and explore the space by sampling those configurations that yield maximum information. We then show that with significantly lower precision in the last layers we achieve a minimal loss of accuracy with appreciable memory savings. We test our findings on the CIFAR10 and ImageNet datasets using the VGG, ResNet and GoogLeNet architectures.

Keywords: Quantization · Bayesian Optimization · Gaussian Process

1 Introduction

The strategy of quantizing neural networks to achieve fast inference has been a popular method of deploying neural networks in compute constrained environments. Its benefits include significant memory savings, improved computational speed, and a decreased cost in the energy needed per inference. Many methods have used this family of strategies, quantizing down to anywhere between 8-bits and 2-bits, with little loss in accuracy [10,30]. It also bears noting that in most of these methods, after quantizing to very low precisions (1 to 5 bits), retraining is necessary to recover accuracy.

Recently, even though the quantization algorithms have significantly improved, they have almost exclusively *implicitly* assumed that the best strategy is to quantize all the layers uniformly with the same precision. However, there are two main reasons to believe otherwise: i) we argue that as it has been interpreted [17] that different layers extract different levels of features, it follows that different layers might require different levels of precision; ii) the idea of quantization as an approximation to the floating point (FP) version of the

© Springer Nature Switzerland AG 2020
A. Vedaldi et al. (Eds.): ECCV 2020, LNCS 12362, pp. 383–398, 2020.
https://doi.org/10.1007/978-3-030-58520-4_23

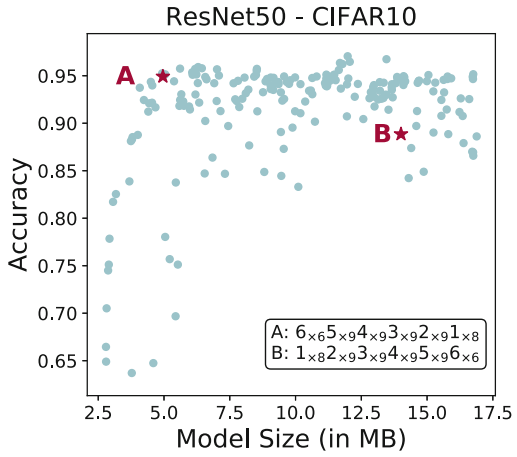


Fig. 1. Gaussian Process prediction for bit distribution in memory vs accuracy plot

network suggests that lower error in the early layers reduces the propagation of errors down the whole network, minimizing any drop in accuracy. We believe that as important as having a good quantization strategy, is to also have a good strategy for the distribution of bits through the network, thereby eliminating any redundant bits. The goal is then to find a configuration in a search space that uses the least amount of bits and achieves the highest accuracy per bit used.

We cast this Neural Architecture Search (NAS) problem into the framework of hyperparameter search, since the bit-width of each layer should ideally be found automatically. As with many NAS approaches, measuring the accuracy of a single configuration can take a considerable amount of time. To mitigate this issue, we propose a two stage approach. First, we map the full search space into a lower dimensional counterpart through a parameterised constraint function, and second, we use a Multi-task Gaussian Process to predict the accuracy at a higher epoch number from lower epoch numbers. This approach allows us to reduce both the complexity of the search space as well as the time required to determine the accuracy of a given configuration. Finally, as our Gaussian Process based approach is suitable for probabalistic inference, we use Bayesian Optimisation (BO) to explore and search the hyperparameter space of variable bit-size configurations.

For the quantization of the network, we use the DSConv method [16]. It achieves high accuracy without significant retraining, meaning the number of epochs needed for full training, and implicitly, the requirement for prediction power, is minimised.

To summarise, our main contributions are as follows:

1. we cast NAS as hyperparameter search, which we apply to the problem of variable bit-size quantization;
2. we reduce the time needed to measure the accuracy of a proposed bit configuration considerably by using multi-task GPs to infer future accuracy from current estimates;
3. we demonstrate performance across a broad range of configurations, described by Bezier curves and Chebyshev series.

The next sections are as follows: Sect. 2 shows previous work on quantization and hyperparameter search. Section 3 elaborates on the methodology used for search, including the constraint, exploration, and sampling procedures. Section 4 shows the results achieved on the CIFAR10 and ImageNet datasets using the networks listed above. Section 5 draws a conclusion and considers insights from the paper.

2 Related Work

Neural Architecture/Hyperparameter Search. One can consider finding bit distributions as a form of model selection [18], given its complexity and the limit on the parameters that it accepts as a solution. Previous methods have predominantly used Reinforcement Learning (RL) and Evolutionary Algorithms (EA) to model search, which is referred to in the literature as Neural Architecture Search. Examples include NASNet [34], MNasNet [26], ReLeq-Net [6], HAQ [29], among others [1,31] for RL and [13,15,24,33] for EA. Our work overlaps with these papers only on the goal of finding an optimal strategy given a search space.

ReLeQ-Net and HAQ, to the best of our knowledge, are the only methods whose aims are to find the optimal bit distribution through different layers of a network, and are therefore the papers that overlap the most with our work. It is notable that both of them use an RL based approach to search for optimal bit distributions. However, HAQ is more focused on hardware specific optimization, whereas both ours and ReLeQ-Net's methods attempt to be Hardware-Agnostic. Recently some work involving Bayesian Optimization (BO) for model architecture selection has been carried out, with systems such as NASBOT [11]. One of the reasons why BO has not been used for model selection has to do with how unclear it is to find a measure of "distance" between two models, which is the main problem that was addressed by NASBOT.

Alternatively, one can see determining bit distribution as finding hyperparameters to be tuned given a model, *i.e.* not different from finding the optimal learning rates or weight decays. Historically, this has been tackled by BO techniques. In neural networks specifically, this was popularized after the work of [21], and followed by others [2,7,22,27]. As a result BO can be considered a natural method for searching for optimum bit distribution configurations.

Quantization. Quantization strategies can be either trained from scratch or derived from a pretrained network. The methods of [4,10,30,32] initialize their networks from scratch. This ensures that there is no initial bias on the values of the parameters, and they can achieve the minimum difference in accuracy when extremely low bit values are used (1-bit/2-bits) - a notable exception being DoReFa-Net [32], which reportedly had slightly better results when quantizing the network starting from a pretrained network. The methods of [8,14,16,28] quantize the network starting from a pretrained network. These methods start with a bias on the values of the parameters, which can limit how much they recover from the lost accuracy. A benefit of these methods though is that they

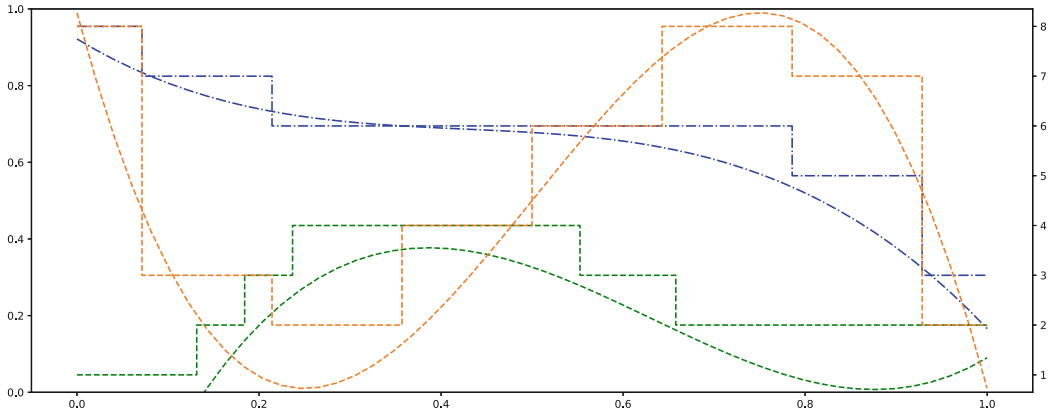


Fig. 2. Three examples of modified Chebyshev functions and their clamped versions. The continuous lines represent the values of the modified Chebyshev functions as function of the layer, which is then converted into bitwidths whose values are represented on the right hand axis. This is for two 8 layer VGG11s and a 20 layer ResNet corresponding to configurations 1) blue: 8,7,6,6,6,6,5,3 2) orange: 8,3,2,4,6,8,7,2 and 3) green: 1,1,1,2,3,4,4,4,4,4,4,3,3,2,2,2,2,2,2,2 (Color figure online)

can be quickly fine-tuned over a few epochs re-achieving state-of-the-art results. These methods are more interesting to us because of their quick deployment cycle. It is worth noting that all of these methods use a uniform distribution of precision, meaning that all layers are quantized to the same number of bits.

3 Method

Our method consists of three parts: constraining, exploring, and sampling the search space. We first constrain the search space by assuming dependence between adjacent bit numbers. We do this by drawing bit distributions from a low-degree Polynomial (in the experiments we use a 2^{nd} degree Bezier curve and a 4^{th} order Chebyshev series). Given a these distributions, we quantize the network using the DSConv [16] method. We explore the space by placing a Gaussian prior over the polynomial parameters, and sampling/retraining a set of hyperparameters that gives the most information about the final payoff function. After exploring, we rank the configurations based on sampling the GP for accuracy, and choose the ones that are the most appropriate for our end-use. Each of these phases will be explained further in this section.

3.1 Constraining the Space

When trying to find the bits, from 1–8, for each layer, the search space will have size of 8^n, where n is the number of layers of the network. For a CNN of 50 layers, the search space will be $2^{150} \approx 10^{45}$, which is a similar size to a game of chess ($\approx 10^{50}$). The size of this space makes an exhaustive search prohibitive.

Our method for constraining the search space relies on the use of parameterised functions. We model a function of degree n with a few parameters, which

describe the search space. We then discretise the function, such that a bit configuration of any layered size network can be sampled from a few parameteres alone.

We use two parameterised functions to illustrate our solution:

- We define the Bezier function $\mathbf{B}(x;\mathbf{w}) = \mathbf{w}^T\phi(x)$ for $x \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d,\ 0 \leq x \leq 1,\ 0 \leq w_i \leq 1\ \forall i \in \{i : i \in \mathbb{N}^+, i <= d\}$, where d is the degree of the polynomial. The vector $\phi(x)$ is the feature vector of the Bezier curve *i.e.* for Linear Bezier $\phi(x) = [1-x,\ x]^T$, for Quadratic Bezier $\phi(x) = [(1-x)^2,\ 2(1-x)x,\ x^2]^T$, *etc.*
- We define the modified Chebysehv function $\mathbf{T}(x;\mathbf{w}) = \frac{(\mathbf{w}-0.5)^T\phi_d(x)+1}{2}$ for $x \in \mathbb{R}, \mathbf{w} \in \mathbb{R}^d,\ -1 \leq x \leq 1,\ 0 \leq w_i \leq 1\ \forall i \in \{i : i \in \mathbb{N}^+, i <= d\}$, where d is the degree of the polynomial. The vector $\phi_d(x)$ is defined as $[T_0(x), T_1(x), T_2(x), \ldots, T_{d-1}(x)]^T$ where $T_0(x) = 1, T_1(x) = x$, and $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$

The constraint function, $g(t)$, is then a clamped and rounded version of the chosen polynomial, $p(t)$, such that the bits, b, for each layer generated are between 1 and 8, and $b_i \in \mathbb{N}$. We can define then $g(t) = \lfloor \text{CLAMP}(8p(t)+1), 1, 8) \rceil$, where $\lfloor \cdot \rceil$ is the rounding function, and $\text{CLAMP}(f, a, b) = \min(\max(f, a), b)$.

Figure 2 shows an example of a Chebyshev function and its clamped version. The y-axis in the left indicate the value of the Chebyshev function for different values of x. This is then clamped, rounded, and scaled such that it transforms into a discontinuous line that represents the bit chosen for each layer of a CNN. The bit value is indicated in the y-axis in the right.

By constraining the search space in this way, the minimization problem then shifts as follows:

$$\begin{array}{ll} \textbf{Naïve Approach} & \\ \min\limits_{\mathbf{b}} & \mathcal{L}(t;\mathbf{b}), \mathbf{b} \in \mathbb{N}^n \\ \text{s.t.} & 1 \leq b_i \leq 8,\ \forall i \in \{1, n\} \end{array} \qquad \begin{array}{ll} \textbf{Our Approach} & \\ \min\limits_{\mathbf{w}} & \mathcal{L}(t;\mathbf{w}), \mathbf{w} \in \mathbb{R}^d \\ \text{s.t.} & b_i = g(\frac{i}{n}),\ \forall i \in \{1, n\} \\ & 0 \leq w_j \leq 1,\ \forall j \in \{1, d\} \end{array} \tag{1}$$

where $\mathcal{L}$ is the loss function (to be introduced in Sect. 3.3).

The search then reduces to finding the parameters of the polynomial basis $\mathbf{w}$, which consequently define the bit distributions throughout the layers. The search space is then continuous and compatible with GPs, and significantly reduced to only d dimensions. Using this parameterisation, we are able to easily define a distance metric between configurations to be used when calculating the kernel function and predictive distribution from our Gaussian Process. So, with this setup, the search space can be sufficiently explored in a timely manner.

Quantization Strategy. The method used for quantizing the CNN is DSConv. This is because our aim is to minimize time taken during training, and DSConv has consistently shown good accuracy properties in models, even before retrained.

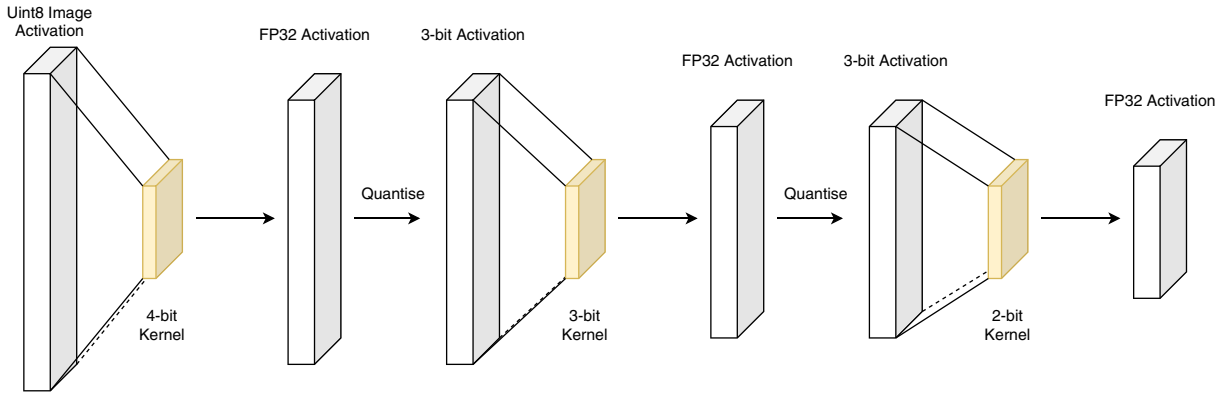


Fig. 3. Quantization given variable bit-widths. Notice that the input is the image, which is a `uint8` tensor (normalization can be dumped into a KDS tensor [16]), so it is not quantized. The quantization of activations is done before the convolution such that the convolution can be done using the same precision.

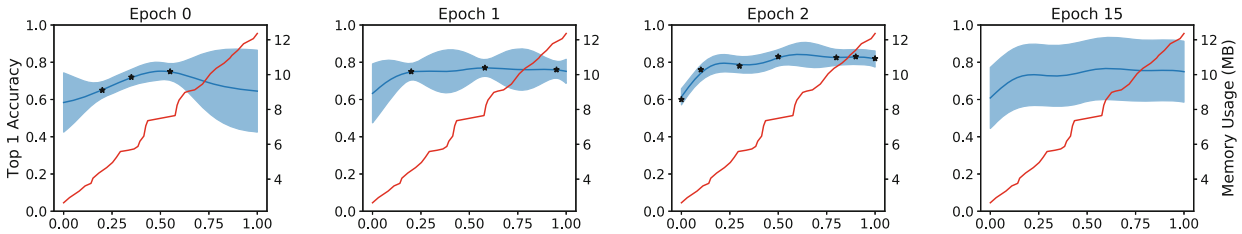


Fig. 4. Multi-task Gaussian Process for inferring accuracy of quantized network. The quantization function is a Bezier Linear with the first parameter set to 0.5 *i.e.* $g(t; w_1) = 0.5 + t(w_1 - 0.5)$. For all figures, the x-axis is the value of w_0, the left y-axis is the accuracy on CIFAR10 of a toy CNN with 10 layers. The right y-axis (red line) shows the model size for a given value of w_0. The epoch correspondences for each task is [0, 1, 2, 15] respectively. After this exploration phase, the decision procedure is run on the predictive distribution of Task 4. (Color figure online)

In this method, both the activations and the weights are quantized, such that fast inference is possible. Each of the weight tensors are divided into blocks of size $B = 32$ depthwise. Each block holds B integers and one FP32 multiplier value. The integers are found by simply scaling each of the block from the original FP32 weight tensor by $\frac{2^{b-1}}{\text{MAX}(w)}$, and then flooring and cropping to range. The FP32 value is calculated by simply minimizing the L2 norm of the block with respect to the original corresponding block: $\xi = \frac{\sum_{i=0}^{B-1} w_i w_{qi}}{\sum_{i=0}^{B-1} w_{qi}^2}$. The activation tensor is quantized similarly, but using Block Floating Point (BFP) format in each of the blocks instead.

In order to take advantage of the low bit multiplication speed, the activation tensor and the weight tensor need to have the same precision. Figure 3 shows how this is done. The activation tensor prior to a convolution layer is set to be quantized to the same bit precision as that layer. The first convolution is not quantized since the input image is already in `uint8` format. Also note that we quantize only the convolutional layers. The Fully Connected layers are all left in the original FP32 precision for training.

3.2 Exploring the Space

Next, we need a way of exploring the space in order to learn the accuracy of the network given a limited set of $\mathbf{w}$ points. We propose a Multi-Task Gaussian Process prior in the neural network, such that each task corresponds to the estimation of the accuracy of the quantized network given $\mathbf{w}$ after a certain number of epochs, *e.g.* task 1 corresponds to 0 epochs, task 2 to 1 epoch, task 3 to 2 epochs, task 4 to 15 epochs. Let there be m tasks, and a prior on f_l, $l \in \{0, m\}$, such that $f_l \sim \mathcal{GP}(\mu(t), k(t, t'))$. We also place a probability distribution $\mathcal{P}(f_0, f_1, ...f_m)$ over different tasks. Let $y_{(t,l)} = f_l(t) + \epsilon(t)$ be the observation at hyperparameter value t for task l, and let $\epsilon(t) \sim \mathcal{N}(0, \sigma^2; t)$ be the observation noise, which is normally distributed. This defines independent Gaussian Likelihoods $y_{(t,l)} \sim \mathcal{N}(f_l, \sigma^2; t)$. From this model, observations $\mathbf{y}$ are drawn, such that $\mathbf{y} = (y_{11}, ..., y_{s1}..., y_{12}, ..., y_{s2}, ..., y_{1m}, ..., y_{sm})$, where y_{il} is the i^{th} observation of the l^{th} task [23].

We used the Intrinsic Correlation Model (ICM) of [5] and [3] for kernel calculation (in our experiments we made use of the squared exponential kernel). We can then define the mean and the correlation between tasks as:

$$\begin{aligned} \langle f_l(x) \rangle &= \mu_l(x) \\ \mathbb{C}(f_l(x), f_{l'}(x')) &= k^f(l, l')k^x(x, x') \end{aligned} \tag{2}$$

where k^f and k^x are positive semi definite functions, corresponding to the correlation between functions and the correlation between inputs respectively. From this it follows that the covariance is $K = K^f \otimes K^x$, where $\otimes$ is the Kronecker product, K^f is the matrix of correlations between the functions and K^x is the matrix of correlations between the inputs. For a new set of data points x_*, the mean prediction can then be calculated using the normal formula for the predictive distribution:

$$\begin{aligned} \bar{\mathbf{f}}(x_*) &= \mu_l(x_*) + (K^{x_*})^T \Sigma^{-1} \mathbf{y} \\ \Sigma &= K + D \otimes I \end{aligned} \tag{3}$$

where D is an $m \times m$ diagonal matrix where the $(l, l)^{th}$ term is σ_l^2.

Figure 4 shows an example of the Multi-Task setting with a 1D Bezier Curve for ease of visualization. Each plot shows the predictive mean and variance for each epoch after 14 data points have been collected, using the exploration algorithm explained in Sect. 3.2. The idea is to predict what is the distribution of the last task given inputs in earlier tasks.

Exploration Phase. In order to make decisions on what parameters to choose, we need to explore the space to predict the accuracy of the last task. The exploration phase for the multitask Gaussian Process follows the Low-Fidelity Search from [23]. The idea is to find the values of x, l such that it gives us maximal information $\mathbb{I}(y_{(x,l)}; f_m |\ \mathbf{y}) = \mathbb{H}(y_{(x,l)} \mid \mathbf{y}) - \mathbb{H}(y_{(x,l)} \mid \mathbf{y}, f_m)$, where $\mathbf{y}$ is the observation history, and (x, l) is the action to be performed. It is important to weight the information by a measure of the cost that it takes to perform that operation.

So the exploration procedure chooses x, l that maximizes $\mathbb{I}(y_{(x,l)}; f_m | \mathbf{y})$ per unit cost. This means that the parameter that has the most information about the payoff function will be picked.

Depending on the dataset and model chosen, the user can favour exploration on one fidelity over the other by decreasing the cost λ of running that particular task. Additionally, we set up a budget on the amount of time in unit cost or number of architectures that we are willing to explore. The Exploration Phase finishes when the Budget has been fully used. After this is finished, the user can run their preferred method of ranking configurations using the posterior of the trained GP.

3.3 Sampling the Space

The naïve goal is to find the highest accuracy per bit possible, which corresponds to finding the minimum of the loss function $\mathcal{L}(t; \mathbf{w}) = -\frac{y_{(t,m)}}{\sum_{i=1}^{n} b_i}$. However, there is a trade-off that must be considered. A model, *e.g.* ResNet20, using a total of 40 bits and achieving 80% accuracy (ratio of 2%/bit) is arguably worse than a model that uses 43 bits and achieves 85% accuracy (ratio of 1.97%/bit). The goal is instead to find a decision procedure that takes into account the regret of not using more bits based on a set of constraints. This relationship should be linear instead of inversely proportional. A better strategy is to assume that using 4-bits for all layers is the lowest uniform quantization scheme without loss of accuracy. Each bit used less than this should be a reward, and each bit used more than this should be a penalty, this is added (or subtracted) to the accuracy to get an "effective accuracy". We then define the effective accuracy as $\mathcal{E}(a, \mathbf{b}, n) = a - \frac{\sum_{i=1}^{n} b_i - 4n}{k}$, where a is the accuracy of the original network, and k is a constant of penalty per bit. Therefore, for $k = 100$ each bit used in addition to the average of 4-bits incurs a penalty of 1% in the effective accuracy. The reverse incurs a reward of 1% in the effective accuracy. The decision procedure becomes then to minimize the negative effective accuracy, $\mathcal{L} = -\mathcal{E}(a, \mathbf{b}, n)$. Once we have enough information about the GPs, we can rank configurations based on their loss in order to pick the most relevant for us.

4 Experiments and Results

We tested our method in a variety of configurations, using versions of the original VGG, ResNet, and GoogLeNet models, altered in order to take CIFAR10, and ImageNet32 as input. For training CIFAR10 and ImageNet32, we used data augmentation by cropping 32×32 image of the 4-pixel padded original. We used a SGD optimiser with momentum of 0.9, and weight decay of 5×10^{-5}. The learning rate started as 10^{-1}, and was divided by 10 after 150 and 250 epochs.

We ran the exploration procedure on ∼65 configurations for each network using the multi-task algorithm outlined above. From these configurations, we could then use the mean of the gaussian to draw estimates of the accuracy

of many different quantization schemes. Using the decision outlined above, we sorted the results by either accuracy, memory, or computational complexity, and selected the points of interest for better visualization and intuition of what the general trend of the found configurations are.

Results on Accuracy Using the CIFAR10 Dataset. Results on CIFAR10 and ablation tests are displayed in Table 1. The configurations are color coded for clarity, with red representing higher bit counts and green representing lower bit counts. These configurations were selected based on the decision procedure outlined above, using the Bezier Linear polynomials.

For comparison, we show 6 configurations of each network: the first and third configurations were picked by our decision procedure; the second and fourth are simply the inverse order of the first and third configurations; the fourth and fifth rows use the traditional uniform distribution of bits for a fair comparison.

It is important to note that the decision to pick these configurations are based on the estimate of the GP rather than on the actual Top-1 results. In order to compare fairly, we also included the Top-1 score and standard deviation from 10 runs after properly training each of them for an additional 30 epochs using the same hyperparameters and optimiser that were used to train their FP32 version. We have also included a delta column which shows the difference between the Top-1 estimate from the GP and the Top-1 after fine-tuning the network. It is remarkable that most of the error in estimation is within 1%, which shows how the GP was able to generalize and interpolate properly as expected.

It can be seen that in general, using more bits in earlier layers yields more accurate, and lighter configurations. The higher accuracy can be explained numerically, since higher bits are used in earlier layers, the error propagation through the network is smaller. The lower memory usage is due to the fact that later layers have a higher number of channels, and therefore using lower precision in those layers yield a massive difference in memory need. For VGG16, the first configuration is both lighter, faster, and more accurate than using 3-bits for all layers. This pattern is repeated for the deeper VGG19 too, where the first configuration yielded superior results to the constant 3-bits for all layers, and also for ResNet18 as well. This "rule of thumb" is somewhat weaker in the GoogLeNet architecture though, even though there is still a clear correlation.

Results on Accuracy Using Chebyshev Series. In order to test robustness of the method in relation to the choice of prior functions, we chose to use a Chebyshev Series of fourth degree, which has a larger search space than the Bezier Linear model. We have tested the model using the CIFAR10 dataset as well, and the results are shown in Table 2.

As it can be seen, the 4^{th} degree introduced more flexibility as to what bit configurations the method is capable of finding. We found that with higher degree of polynomials, the number of architectures to search should also increase. In our experiments, we have searched for $\sim$150 configurations before finding good results. The table shows the expected result that more bits at the beginning compensate for the fewer bits at the end of the network. The ResNet-18 result

Table 1. Results for many configurations on CIFAR10. VGG16 and VGG19 correspond to the architectures introduced in [20]. ResNet18 is the architecture from [9], and the GoogLeNet architecture is from [25]. The Configuration refers to the bit value for each layer of a given model, from earlier layers in the left to later layers in the right. They are color coded for clarity: red for higher bits and green for lower bits. It is important to note that we quantize only the convolutional layers, which means that VGG16 has 13 values, VGG19 has 16 values, ResNet18 has 20 values, and GoogLeNet has 64. Because of its size, the GoogLeNet values were represented by a subscript indicating the number of times that a given bit-width is used. The column "Delta" refers to the difference between the GP estimation of the Top1 accuracy and the actual mean Top1 accuracy ($n = 10$) after properly retraining that particular configuration.

CNN	Configuration (bits per layer)	Top 1 Estimate from GP	Mean Top1	Std	Delta	# Bits	Memory (in MB)
VGG16	32-bit Floating Point	–	93.7%	–	–	–	58.8
	6 555 44 333 22 11	(95.5%)	**93.7%**	0.2%	−1.8%	50	**4.84**
	11 22 333 44 555 6	(91.3%)	**87.7%**	0.2%	−3.6%	50	**9.50**
	7 66 55 44 33 222 1	(92.1%)	93.7%	0.1%	1.6%	50	5.26
	1 222 33 44 55 66 7	(90.1%)	91.5%	0.4%	1.4%	50	10.75
	4444444444444	(93.3%)	93.8%	0.1%	0.5%	52	8.28
	3333333333333	(92.9%)	93.5%	0.2%	0.6%	39	6.44
VGG19	32-bit Floating Point	–	93.9%	–	–	–	80.1
	6 555 4444 333 222 11	(94.4%)	**93.7%**	0.1%	−0.7%	54	**6.95**
	11 222 333 4444 555 6	(91.6%)	**89.6%**	0.4%	−2.0%	54	**12.04**
	5 4444 33333 2222 11	(93.9%)	93.5%	0.1%	−0.4%	46	6.14
	11 2222 33333 4444 5	(90.3%)	88.4%	1.2%	−0.9%	46	10.05
	3333333333333333	(92.9%)	93.4%	0.2%	0.5%	48	8.76
	2222222222222222	(92.1%)	92.2%	0.2%	0.1%	32	6.25
ResNet18	32-bit Floating Point	–	95.4%	–	–	–	44.6
	666 5555 4444 3333 2222 1	(96.3%)	**95.4%**	0.1%	−0.9%	75	**3.72**
	1 2222 3333 4444 5555 666	(95.9%)	**92.9%**	0.3%	−3.0%	75	**8.00**
	44444444 3333333333 22	(95.3%)	95.3%	0.1%	0.0%	60	4.34
	22 3333333333 44444444	(94.5%)	94.2%	0.2%	−0.3%	66	6.08
	33333333333333333333	(94.4%)	95.0%	0.1%	0.6%	60	4.90
	22222222222222222222	(93.1%)	93.3%	0.5%	0.2%	40	3.49
GoogLeNet	32-bit Floating Point	–	95.5%	–	–	–	24.32
	$4_{\times 21}$ $3_{\times 27}$ $2_{\times 16}$	(94.7%)	**95.3%**	0.1%	0.6%	207	**2.35**
	$2_{\times 16}$ $3_{\times 27}$ $4_{\times 21}$	(94.6%)	**94.2%**	0.1%	−0.4%	207	**2.98**
	$6_{\times 8}$ $5_{\times 13}$ $4_{\times 12}$ $3_{\times 13}$ $2_{\times 13}$ $1_{\times 5}$	(95.8%)	95.3%	0.1%	−0.5%	231	2.65
	$1_{\times 5}$ $2_{\times 13}$ $3_{\times 13}$ $4_{\times 12}$ $5_{\times 14}$ $6_{\times 8}$	(94.7%)	90.5%	0.2%	−4.2%	231	3.73
	$3_{\times 64}$	(94.7%)	95.1%	0.1%	0.4%	192	2.68
	$2_{\times 64}$	(93.4%)	93.5%	0.2%	0.1%	127	1.92

resembles the configuration found in Table 4, even though it found a configuration that has more usage of 3-bits, but performs slightly worse. As also expected, when the bit distribution is inverted in the network, it results in both higher memory and lower accuracy.

The same behaviour is found with the VGGs, with the slight difference that as VGG11 is too shallow, it requires more bits to recover the accuracy. VGG16 is considerably deeper, and therefore our algorithm was able to compress it more significantly.

This results shows that our method can be used with a variety of basis. It is worth bearing in mind that the GP processing capability requires the inversion of a matrix, which is proportional to the degree of the polynomial chosen. Therefore our method will only work in a timely manner when using fewer hyperparameters to describe the function.

Table 2. Results of method when using Chebyshev Polynomials of 4^{th} degree.

Method	Network	Bitwidths	Accuracy loss	Memory (as a % of original)
Ours	ResNet18	6 4 33 222 33333333 22	−0.6%	8.0%
		22 33333333 222 33 4 6	−1.2%	11.5%
Ours	VGG11	7 666 7 6 5 4	−0.1%	16.8%
		4 5 6 7 666 7	−0.5%	19.7%
Ours	VGG16	4 3 222 3333 22 11	−0.8%	6.3%
		11 22 3333 222 3 4	−2.4%	8.3%

Results on Network Size. Figure 5 shows the result of the GP-estimated accuracy of different configurations by their model size. The solid purple line links the uniform configurations, starting with all 1s and finishing with all 6s. Therefore, any point that lies above that line is an interesting point, since it gives better accuracy by using the same amount of memory of its uniform counterpart. We have highlighted a number of different interesting configurations with red stars and labelled them from A-M in order to better visualise what each point represent.

As it can be seen in the figure, the choice of bit-usage throughout the network plays an important role in both the accuracy and the memory usage. Even though there is a clear trend that links model size and accuracy, there are a handful of configurations which can perform well on both fronts. It can be seen that, in general, points that are above the purple line are linearly decreasing with bit-usage whereas the ones that are below the purple line are linearly increasing with bit-usage.

The surprising result is that, in the CIFAR10 experiments, even though using uniformly 1-bit for all layers achieves bad results, by just introducing a couple of bits in the first three quarters of the network (such as in points A, C and F), the memory increase is almost negligible, but the accuracy recovery is significant. Adding bits at the end of the network however, achieves the opposite effect. It can also be noticed that points A, C, E, and F, achieve better accuracy than the uniformly 2s configuration whilst using 50% less memory. This is even more evident in point E, in which we used up to 6 bits in the first layers, but still achieved less memory usage due to the usage of 1-bit in the bigger kernels at the end of the network.

In the ImageNet32 experiments, we also see some improvement, albeit less dramatic than the CIFAR10 experiments. The overall massage is still the same, as it can be seen in points H, J and L, for which adding bits in the first layers has achieved good accuracy with small memory increases. It is still noteworthy that even with a dataset as challenging as ImageNet32, due to its substantial decrease of information when compared to the default ImageNet, the GP could find good configurations without needing more datapoints. This shows that this method can be robust to changes in dataset.

Brief Comparison with ReLeQ. One of the other papers that touched in this subject was ReLeQ [6]. As explained in the literature review section, they use a reinforcement learning approach to find optimum bit-distributions over the network. Whilst their quantization methodology varies greatly from that used in this paper, it is worth comparing their results to ours. Their results for the CIFAR10 dataset in two of the networks are shown in Table 3. It can be seen that we achieve similar results for ResNet, though with different mean bits. Since ReLeQ's method does not use the same constraint as our method, it could find more varied solutions. This is a limitation to our method which allows it to find solutions to the network faster whilst using less computational power, but reduces the freedom of choice.

Table 3. Comparison of our accuracy results with ReLeQ's method [6] on CIFAR10. The authors in [6] did not provide their models size in memory, so we estimated using both our and the original author's quantisation scheme to make a fair comparison.

Method	Network	Bitwidths	Accuracy loss	Model size (MB)	
				DSConv	WRPNx1
ReLeQ [6]	ResNet-20	8 22 3 222 3 2 333 222 3 2222 8	0.12%	3.88	3.25
Ours	ResNet-20	666 5555 4444 3333 2222 1	0.1%	3.54	2.91
ReLeQ [6]	VGG-11	8 5 8 5 6666 8	0.17%	6.86	6.61
Ours	VGG-11	777 666 55	0.14%	6.35	5.42
ReLeQ [6]	VGG-16	888 6 8 6 8 6 8 6 8 6 8 6 88	0.1%	13.32	12.54
Ours	VGG16	6 555 44 333 22 11	0.1%	4.62	3.74

Results on ImageNet Using ResNet. For completeness, we have included some of the results found by our algorithm on the more challenging ImageNet dataset [19]. This was trained using an Adam Optimizer [12], with learning rate of 10^{-5}.

Table 4 shows the results. As expected, the same pattern of decreasing precision downstream holds across datasets. Comparing these results with the results from DSConv [16], we can see that a decreasing bit-width throughout the architecture, decreasing from 6 bits to 2, is superior to the "all 4s" and "all 3s".

As with ReLeQ's method, HAQ's method has a weaker constraint on bit distribution, which means it would be able to find configurations that our method would not; However, even with our very strong constraint, we were still able to find configurations that are competitive in memory requirements to those found by HAQ. This shows the strength of the conclusion that later layers require lower precision than earlier layers to maintain the same accuracy.

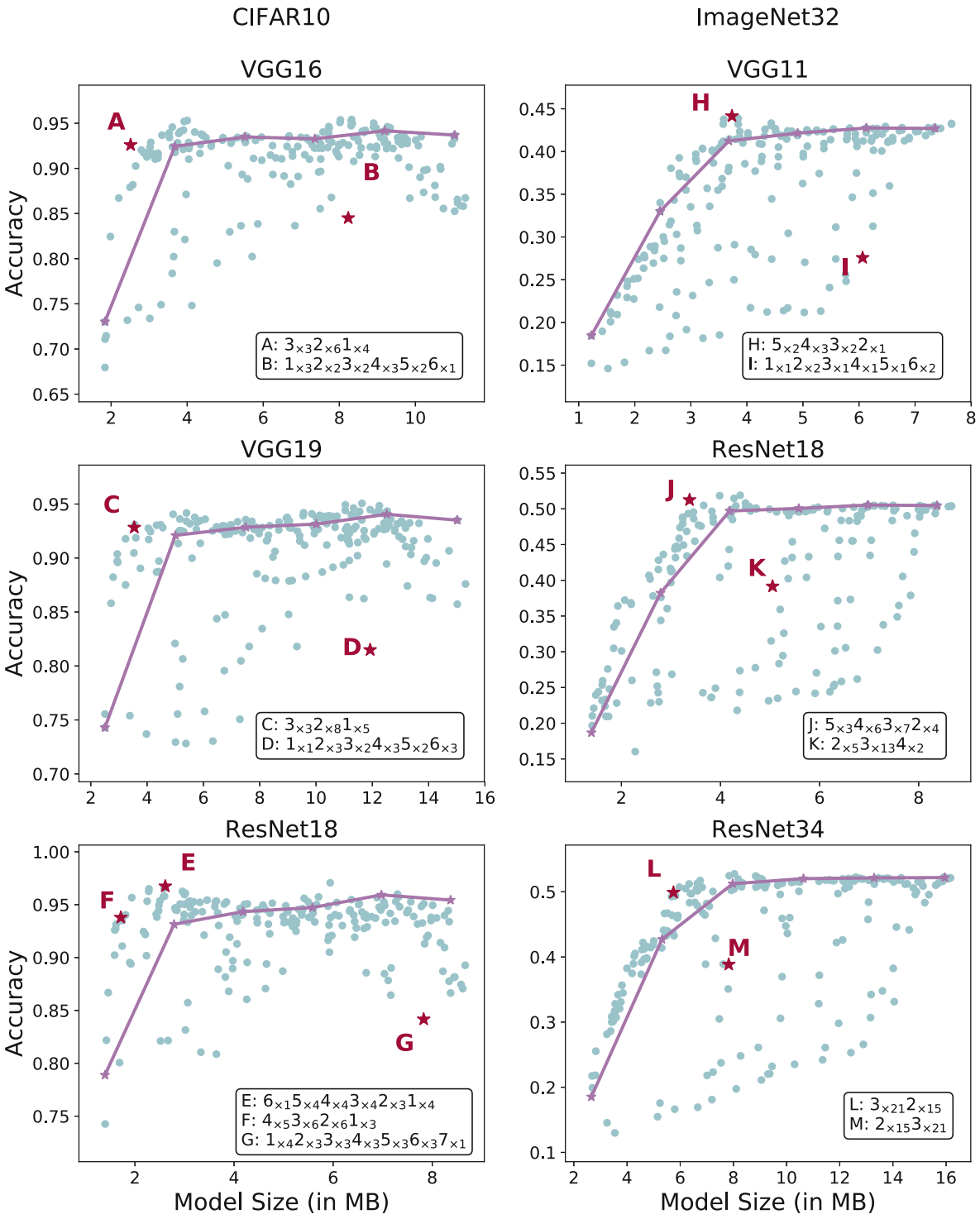


Fig. 5. Scatter plot of the effect on accuracy versus model size of different bit configurations. The left three plots use the CIFAR10 dataset and the right three plots use the ImageNet32 dataset. Note that this is the plot of the estimate as given by the trained GP, and not the actual accuracy given proper training. The solid line refers to the uniform configurations, starting with all 1s and ending with all 6s. Points A-M highlight different configurations as shown in the text boxes. The string of numbers shown refers to the bit size on each layer of the given network. Note that VGG11 has 8 convolutional layers, and therefore points A and B have only 8 numbers. This applies to VGG16 (13 layers), VGG19 (16 layers), ResNet18 (20 layers), and ResNet34 (36 layers) as well.

Table 4. Results of our method using the ImageNet dataset with the ResNet architecture.

Method	# of Layers	Bitwidths	Acc. Loss	Size (MB)
Ours	18	6 ×3 5 ×5 4 ×5 3 ×5 2 ×2	0.2%	**4.89**
DSConv [16]	18	4	0.0%	5.88
DSConv [16]	18	3	0.8%	4.55
Ours	50	6 ×6 5 ×15 4 ×14 3 ×15 2 ×3	0.6%	**11.89**
DSConv [16]	50	4	0.0%	14.54
DSConv [16]	50	3	0.9%	11.74
HAQ [29]	50	*flexible*	0.0%	12.14

5 Conclusion

In this paper, we demonstrate that a uniform distribution over bit-widths throughout a CNN is likely not the most efficient way to quantize a neural network. In order to demonstrate this, we used a Multi-Task Gaussian Process prior over different training epochs, and a Bayesian Optimization exploration procedure based on Information Maximization that estimated the accuracy of different configurations.

We have observed that starting a CNN with higher bit-widths and decreasing precision in later layers yield better accuracy and better memory usage than the traditional uniformly distributed bit-width. This can be interpreted either numerically (as in less error being propagate down the network), or can be interpreted as the functionality of each layer in the network (earlier layers are concerned with feature extraction and later layers are concerned with classification).

Acknowledgements. This research was supported by Intel and the EPSRC, and we thank our colleagues from the Programmable Solutions Group who greatly assisted in this work.

References

1. Baker, B., Gupta, O., Naik, N., Raskar, R.: Designing neural network architectures using reinforcement learning. arXiv preprint arXiv:1611.02167 (2016)
2. Bergstra, J., Yamins, D., Cox, D.D.: Making a science of model search: hyperparameter optimization in hundreds of dimensions for vision architectures. In: Proceedings of the 30th International Conference on Machine Learning, vol. 28, pp. I–115. JMLR.org (2013)
3. Bonilla, E.V., Chai, K.M., Williams, C.: Multi-task gaussian process prediction. In: Advances in Neural Information Processing Systems, pp. 153–160 (2008)
4. Cai, Z., He, X., Sun, J., Vasconcelos, N.: Deep learning with low precision by half-wave gaussian quantization. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 2017. https://doi.org/10.1109/cvpr.2017.574

5. Chai, K.M.: Multi-task learning with Gaussian processes. Ph.D. thesis, The University of Edinburgh (2010)
6. Elthakeb, A.T., Pilligundla, P., Yazdanbakhsh, A., Kinzer, S., Esmaeilzadeh, H.: ReLeQ: a reinforcement learning approach for deep quantization of neural networks. arXiv preprint arXiv:1811.01704 (2018)
7. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems, vol. 28, pp. 2962–2970 (2015)
8. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. arXiv preprint arXiv:1510.00149 (2015)
9. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
10. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: training neural networks with low precision weights and activations. J. Mach. Learn. Res. **18**(1), 6869–6898 (2017)
11. Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., Xing, E.P.: Neural architecture search with Bayesian optimisation and optimal transport. In: Advances in Neural Information Processing Systems, pp. 2016–2025 (2018)
12. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
13. Kitano, H.: Designing neural networks using genetic algorithms with graph generation system. Complex Syst. **4**(4), 461–476 (1990)
14. Lin, X., Zhao, C., Pan, W.: Towards accurate binary convolutional neural network. In: Advances in Neural Information Processing Systems, vol. 30, pp. 345–353 (2017)
15. Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K.: Hierarchical representations for efficient architecture search. arXiv preprint arXiv:1711.00436 (2017)
16. Nascimento, M.G.D., Fawcett, R., Prisacariu, V.A.: DSConv: efficient convolution operator. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 5148–5157 (2019)
17. Olah, C., Mordvintsev, A., Schubert, L.: Feature visualization. Distill (2017). https://doi.org/10.23915/distill.00007. https://distill.pub/2017/feature-visualization
18. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press (2005)
19. Russakovsky, O., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. (IJCV) **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
20. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
21. Snoek, J., Larochelle, H., Adams, R.P.: Practical Bayesian optimization of machine learning algorithms. In: Advances in Neural Information Processing Systems, pp. 2951–2959 (2012)
22. Snoek, J., et al.: Scalable Bayesian optimization using deep neural networks. In: International Conference on Machine Learning, pp. 2171–2180 (2015)
23. Song, J., Chen, Y., Yue, Y.: A general framework for multi-fidelity Bayesian optimization with Gaussian processes. In: The 22nd International Conference on Artificial Intelligence and Statistics, pp. 3158–3167 (2019)

24. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002)
25. Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
26. Tan, M., et al.: MnasNet: platform-aware neural architecture search for mobile. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2820–2828 (2019)
27. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD 2013, pp. 847–855. ACM, New York (2013). https://doi.org/10.1145/2487575.2487629
28. Vanhoucke, V., Senior, A., Mao, M.Z.: Improving the speed of neural networks on CPUs. In: Deep Learning and Unsupervised Feature Learning Workshop, NIPS. Citeseer (2011)
29. Wang, K., Liu, Z., Lin, Y., Lin, J., Han, S.: HAQ: hardware-aware automated quantization with mixed precision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 8612–8620 (2019)
30. Zhang, D., Yang, J., Ye, D., Hua, G.: LQ-Nets: learned quantization for highly accurate and compact deep neural networks. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds.) ECCV 2018. LNCS, vol. 11212, pp. 373–390. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01237-3-23
31. Zhong, Z., Yan, J., Wu, W., Shao, J., Liu, C.L.: Practical block-wise neural network architecture generation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2423–2432 (2018)
32. Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., Zou, Y.: DoReFa-Net: training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160 (2016)
33. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)
34. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 8697–8710 (2018)

Statement of Authorship for joint/multi-authored papers for PGR thesis

To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Processes
Publication Status	X Published □ Accepted for Publication □ Submitted for Publication □ Unpublished and unsubmitted work written in a manuscript style
Publication Details	*Gennari do Nascimento, Marcelo, Theo W. Costain, and Victor Adrian Prisacariu. "Finding non-uniform quantization schemes using multi-task gaussian processes." In European Conference on Computer Vision, pp. 383-398. Springer, Cham, 2020.*

Student Confirmation

Student Name:	Marcelo Gennari do Nascimento		
Contribution to the Paper	• Conception of the ideas, with contributions from Theo Costain • Designed, and wrote the code, with contributions from Theo Costain • Performed the experiments in collaboration with Theo Costain • Co-created all of the tables and diagrams with Theo Costain • Co-wrote the manuscript with Theo Costain • Prepared and presented the poster session at the conference		
		Date	12/09/2022

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Victor Adrian Prisacariu		
Supervisor comments The description on contributions is accurate		
Signature 	Date	12/09/2022

This completed form should be included in the thesis, at the end of the relevant chapter.

4.5 Conclusion

In this chapter we presented *Finding Non-Uniform Quantization Scheme using Multi-Task Gaussian Process*, which is a method to find optimal bitwidth distributions across the layers of a CNN. The method consists of 3 steps: first constraining the search space by restricting the bit distributions to follow a parameterized function; second exploring the space, whereby the parameters of the function are sampled based on the maximization of the acquisition function of the multi-task gaussian process (GP); third by sampling the space by using the prediction of the now trained GP to find optimal bitwidths according to the trade-offs set by the designer.

Our method was efficient enough to find good configurations by using only 65 sampled distributions. We tested our method on both ImageNet and CIFAR10, using the main deep neural networks such as VGG, ResNet and GoogLeNet. As a rule of thumb, the results showed how in order to minimize memory usage, the best strategy is to use higher precision in the early stages of the network and lower precision in the latter stages.

5

HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation

Contents

5.1 Context 66
5.2 A Primer on Backpropagation in Convolutional Layers 66
5.3 Author Contribution 68
5.4 Publication: Gennari do Nascimento *et. al.* *"HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation" In Prep.* 68
5.5 Conclusion 83

In this Chapter, we will mainly present our work "HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation", which is under review for the Winter Conference on Applications for Computer Vision 2023. We will start by giving some context on the motivation of this work, and what problems it aims solve. We also provide a quick synopsis of backpropagation for the convolutional layer. Then we present our paper, and conclude with an overview of what has been achieved, and the contributions to the community.

5.1 Context

Most methods on quantization focus on inference only. This includes our two previous papers, introduced in Chapters 3 and Chapter 4. However it is still useful to have a method that also works seamlessly for training, since the backward pass of a neural network can take more than 2x the time and energy needed than the forward pass.

Even more useful is a method that uses the same quantization scheme and algorithm for both inference and training: this way only one hardware architecture is needed. The challenge with this approach is that gradients, weight and activations have completely different characteristics. For example, gradients span a larger set of values, and as expected they get closer and closer to zero as training progresses; weights on the other hand generally stay closer to their initialized values and are static; feature maps have a high sparsity because of activation functions such as ReLU and are dynamic.

Our goal is to unify inference and training in one scheme which can be used in a single hardware. In this paper, we explored generalising the "DSConv" quantization algorithm to the backward pass.

5.2 A Primer on Backpropagation in Convolutional Layers

The main problem that our method explores is that using our quantization scheme directly for backpropagation is not feasible due to how the backpropagation is calculated in CNNs. Figure 5.1 shows the operations needed in the backward pass for the convolution operator.

The forward pass works as expected: the only computation needed is the convolution operator between the weight W and the activation A, resulting in the feature map F. In order to perform the backward pass, there are two convolution operators needed: one of them to caluclate the gradient of the loss with respect to

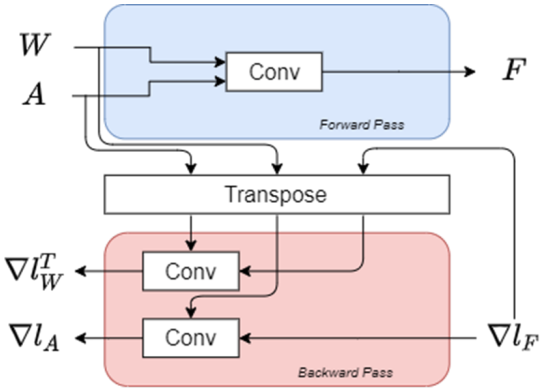


Figure 5.1: Illustration of operations needed in order to perform backpropagation in convolutional layer.

the weights ∇l_W^T, which is used to update the value of W; and the gradient of the loss with respect to the activation ∇l_A^T, which is fed back to the previous layer.

Notice that these are not normal convolutions though: they are transposed convolutions. So before the tensors are fed into the convolutions, some of them need to be transposed. This causes a problem when the tensors are quantized using DSConv, which is illustrated in Figure 5.2.

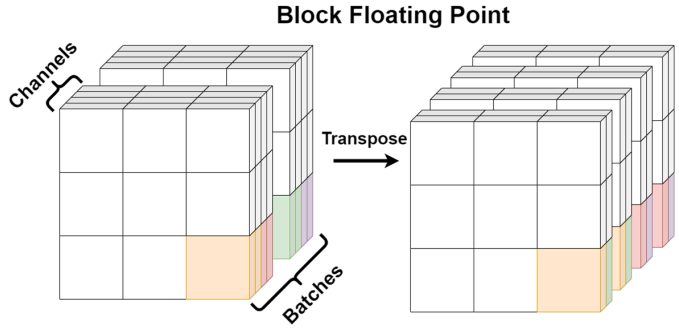


Figure 5.2: Illustration of exponent misalignment caused when a block floating point tensor is transposed. Same colour indicates values that share the same exponent.

In the figure, values of equal colours correspond to exponents that are shared. In this case, we are looking at a tensor with spatial dimensions 3x3, with 4 channels and batch size equal to 2. The block size is also equal to 2. Notice that after the transposition between the batch dimension and the channel dimension, the values that used to share exponents are now in different batches, such that the property of performing the dot product in integer operations cannot be exploited anymore.

On top of this "transposition" problem, another issue that arises when quantizing the backward pass is that gradients get closer and closer to zero, which means that they often get quantized down to zero. If they are repeatedly quantized down to zero, then the weights are not updated, and the network stops "learning". One of the most important mechanisms to avoid the problem of gradients getting stuck at zero is *stochastic rounding*, which was introduced in Chapter 2. We expand their definition and background in the paper, and show some proofs and experiments in the supplementary material of our paper.

5.3 Author Contribution

- A method that uses the same quantization scheme for both inference and training, allowing for only one hardware to be used for both tasks
- An extension to the channel-wise block floating point format which allows for backpropagation to be performed without needing multiple memory accesses
- Extensive results in large datasets and a wide range of neural network architectures

5.4 Publication: Gennari do Nascimento *et. al.* *"HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation" In Prep.*

HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation

Marcelo Gennari do Nascimento
University of Oxford - Active Vision Lab
Parks Road, Oxford OX1 3PJ
marcelo@robots.ox.ac.uk

Victor Adrian Prisacariu
University of Oxford - Active Vision Lab
Parks Road, Oxford OX1 3PJ
victor@robots.ox.ac.uk

Roger Fawcett
Intel Corporation
Eclipse, Globe Park, Marlow SL7 1YL
roger.fawcett@intel.com

Martin Langhammer
Intel Corporation
Eclipse, Globe Park, Marlow SL7 1YL
martin.langhammer@intel.com

Abstract

Prior quantization methods focus on producing networks for fast and lightweight inference. However, in these works the cost of unquantised training is overlooked, despite it requiring significantly more time and energy than in inference. We present a method for quantizing convolutional neural networks for efficient training. Quantizing gradients is challenging because it requires higher granularity and their values span a wider range than the weight and feature maps. We propose an extension of the Channel-wise Block Floating Point format that allows for quick gradient computation, using a minimal amount of quantization time. This is achieved through sharing an exponent across both depth and batch dimensions in order to quantize tensors once and reuse them for the transposed convolution during backpropagation. We test our method using standard models such as AlexNet, VGG, and ResNet, on the CIFAR10, SVHN and ImageNet datasets. We show no loss of accuracy when quantizing AlexNet weights, activations and gradients to only 4 bits training ImageNet.

1. Introduction

Convolutional Neural Networks have become ubiquitous in a variety of applications, from Image Recognition [18] to Face Identification [34]. One drawback of these models is the amount of computational resources required. It has been estimated that in order to achieve the current 11.5% error benchmark on ImageNet, around 10^6 $USD has been spent on computation cost alone [36]. Therefore, significant amounts of research has been conducted in order to make models run quicker, and using less memory and energy.

Quantization is now an important way of addressing this problem. This method has been successfully used by the industry to deploy to embedded devices [14] and servers [8]. However, most of the effort has been put into quantizing models for inference, focusing only on the forward pass. There is significantly less research on gradient quantization.

A prohibitive amount of time is used in the development cycle to train networks, and during the training cycle, the backward pass requires twice as many convolution calculations as the forward pass. Therefore, it is paramount to accelerate the backward pass.

Many factors contribute to this, including the requirement for higher precision for gradient calculations, and the focus on deploying to the end user rather than optimizing for production. Moreover, many methods use different quantization schemes (hardware) for the forward and backward pass. For example, Habana Labs uses a specific hardware for training (Gaudi [19]), and another for inference (Goya [20]); previous papers like DoReFa-Net [49] use different quantization schemes for gradients and weights/ activations.

Building on this motivation, we propose a method to address the main challenges when it comes to calculating gradients using low precision tensors, which uses the same quantization building block both at training and inference. Our method quantizes both the forward and backward passes of a convolutional neural network, such that the bulk of the multiplications and additions are done in narrow precision. Our contributions are as follows:

- A method for quantizing forward and backward passes using the same hardware and quantization scheme.
- An extension of block floating point which allows for less quantization conversion operations.

- Robust results in a variety of datasets and different families of architectures, including no loss of accuracy when quantizing AlexNet down to 4 bits.

Our method employs the Block Floating Point (BFP) method to backpropagation, which can achieve high accuracy using narrow bitwidths by using simple heuristics for exponent sharing. Furthermore, we introduce a variant of BFP that we call HyperBFP, which spares redundant quantization passes to tensors and makes the system simpler to design, requiring lower memory bandwidth.

Using these methods, we were able to achieve high quality results even on the most challenging datasets such as ImageNet. We show that when small block sizes are used, such as 8 or 16, we can get little loss of accuracy when quantizing to as low as 2 bits. When we use slightly bigger block sizes such as 32, we achieve no loss of accuracy even when quantizing weights, activations and gradients to 4 bits.

2. Related Work

Many approaches have been tried to accelerate neural network computation. The most natural way is to select better models. The goal is to find architectures that run quicker than traditional methods on retail hardware, such as CPUs and GPUs. This resulted on finding efficient architectures, both human-designed such as MobileNetV2 [31], ShuffleNet [47], and automated methods using Neural Architecture Search (NAS) such as MNasNet [35] and FBNets [40]. Quantization methods can be used in conjunction with better model selection to get additional speed and energy benefits. This has been shown to work even on the highly efficient MobileNet [15]. There is an extensive literature in quantization, which can be divided by two tasks: Quantized Inference, and Quantized Training.

2.1. Quantized Inference

The goal is to quantize both weights and activations, allowing for quick inference algorithms, since using compatible hardware operations in lower precision are quicker and less spacious than regular FP32 format. Most of these methods assume that the training can be performed using FP32 format, but the inference is restricted to low bitwidths. An example of this type of method include QNN [11], which was one of the first methods to include binary weights. Other examples include Halfwave-Gaussian Quantization (HWGQ [2]), ABC-Net [24], LQ-Nets [44], and PSGD [16]. The work of [22] substitutes the ubiquitously used Straight-Through Estimator (STE) with an adaptive gradient scaling function that shrinks the error given when using STE. Another variation of quantized inference is to quantize models without the need for (unlabelled) data, or using synthetic data, which is referred to as post-training quantization. For example [23,25,26,46] use only a small amount of (unlabelled) data (or synthetic data) to quantize a network. The advantage of this method is that no further training is required, often utilizing only a calibration step.

2.2. Quantized Training

The absolute majority of the quantization methods focus on quantizing the forward pass (inference) part of the model only. There are a few methods though that are able to also quantize the gradients to low bitwidth [41,43]. This also means that the backward pass can be done quicker in specialized hardware. Some of these methods include [45], which uses a fixed-point only computation for forward and backward-pass, employing different bitwidths for different layers (from int8 for weights and activations, to a combination of int8, int16 and int24 for gradients); [6] uses an end-to-end block floating point inspired quantization technique to train CNNs with forward and backward passes quantized; [1] also shows good results when quantizing using 8 bits. DoReFa-Net [49], which aims to achieve binary weights, activations and gradient quantization by using hand-designed transformation functions.

The papers that are most related to our work include SWALP [42], and [48], both of which are able to successfully quantize both the backward and forward pass of convolutional neural networks. Other methods such as ALQ/AMQ [7] and TRN [39] quantize only the stochastic gradient for communication efficiency.

Aside from the academic papers on quantization, the industry has adopted different floating point standards in order to accelerate computation using narrow precision. NVidia uses the TensorFloat32 format on the A100 GPUs [29] which uses 8 bit exponents and 10 bit mantissa; and Google uses the BFloat16 format on the TPU v3 [38], which uses 8 bits exponents and 7 bits mantissa. 8-bit floating point numbers has also being proposed for training [37].

This paper will focus on quantized training, and we will expand the literature by showing a new method of quantizing gradients based on the BFP format.

3. Preamble

The Single Floating Point (FP) format, defined by IEEE-754 [12], consists of representing a number using 32 bits (FP32): 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its decimal value can be calculated by the formula $a = (-1)^{b_0} \times 2^{((b_1 b_2 ...)_2 - 127)} \times 1.b_9 b_{11} ... b_{31}$, where $()_2$ indicates binary format and b_n is the n^{th} binary value of this representation: b_0 fixes the sign, $b_{1...8}$ specifies the exponent, and $b_{9...31}$ specifies its mantissa.

Different formats balance range and precision by changing the number of bits used for the exponent and the mantissa (see BFloat16 [38], TensorFloat32 [29], and FP8 [37]). Fixed point formats are slightly more limited as they lack

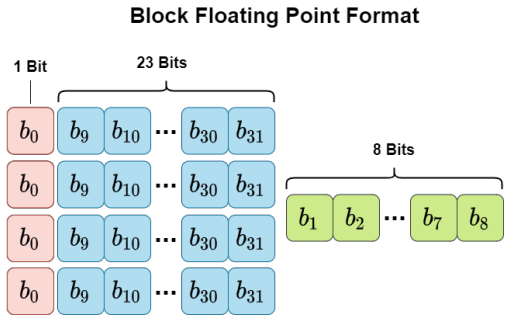


Figure 1: Diagram depicting Block Floating Point format for a block size of 4. Note that in the BFP format, the sign + mantissa bits are fixed-point integer, and the shared exponent acts as a scale value across the entire block.

the possibility of a large range of values, but also require smaller area for operations in the hardware.

The BFP format, rather than being a representation for a single value, consists of a format for a group of values. The representation is similar to FP32, but instead of having one exponent for each number, it shares the same exponent across a certain number of values. Figure 1 shows how it works. Instead of each number being represented by an individual exponent and mantissa values, BFP *shares* a single exponent across in this case 4 values, which are represented then in *fixed-point*. BFP then balances the range of floating point with the efficient computation of fixed point.

This strategy has been applied to CNNs and tensors before. Previous papers have used different strategies for sharing exponents: for inference only [5, 27, 28], and for training as well [4, 6, 32, 42]. Most of the methods share the exponent across entire tensors, or batches. SWALP [42] uses both *"Big-block"* design (which shares one exponent per batched tensor) and *"Small-block"* design, which also shares exponents across batch normalization layers, and fully connected (FC) layers. We believe that these options are unnecessarily restrictive, and more accuracy can be obtained by using smaller blocks. We follow the strategy of [27], in which exponents are shared *channel-wise*. This allows for a fine-grained trade-off between accuracy and computation, whilst taking advantage of fixed point operations in hardware. When converting from FP32 to BFP format, across the channel dimension, for each vector of size blk, the maximum exponent is chosen and it is shared among all other values. This is done by calculating the shift of the mantissas needed for each value to enforce the same exponent.

During inference, when this format is used, the convolution can be performed by using only integer Multiply and Accumulate (MAC) operations per block (channel-wise), plus a small number of FP summations of the resulting scalars, which was shown to be effective for inference.

4. Method

We propose a method for quantizing weights, activations, and gradients for computation in low-precision based on the BFP format. BFP is not invariant to transposition in the input / output channel dimensions (see Figure 2), which complicates convolution operations in the backward pass.

Given a kernel $\mathbf{W}^{(o,i,k,k)}$, input $\mathbf{A}^{(b,i,w_a,h_a)}$, and feature map $\mathbf{F}^{(b,o,w_a,h_a)}$, where i is the input channels, o is the output channels, w_a is the width, h_a is the height, b is the batch size, and k is the kernel width and height, the forward pass of the convolution is defined as:

$$f_{b,o,w_a,h_a} = \sum_{k}^{3}\sum_{k}^{3}\underbrace{\sum_{i_d}^{i} w_{o,i_d,k,k} \cdot a_{b,i_d,w_a+k,h_a+k}}_{\text{Inner MAC}}$$

assuming kernel size equals to three, padding and stride adjusted to preserve the same width and height.

The convolution can be seen as transforming the input features from having i channels, to the output features having o channels. For BFP, the weight and activation share the exponent in blocks, allowing for low-precision computation. Therefore, the Inner MAC in the right hand side of the equation above becomes:

$$\underbrace{\sum_{n}^{\#blk}\sum_{i_b}^{blk} w_{o,n\cdot blk+i_b,k,k} \cdot a_{b,n\cdot blk+i_b,w_a+k,h_a+k}}_{\text{Done in low-precision}}$$

where $\#blk$ indicates the number of blocks depthwise in the convolution (for example, if $blk = 32$ and $i = 128$, then the number of blocks is 4).

For backpropagation to work, both the gradient of the loss with respect to the kernel $\nabla l_{\mathbf{W}} \in \mathbb{R}^{(o,i,k,k)}$, and with respect to the input features $\nabla l_{\mathbf{A}} \in \mathbb{R}^{(b,i,w_a,h_a)}$, given the gradient with respect to the feature map $\nabla l_{\mathbf{F}} \in \mathbb{R}^{(b,o,w_a,h_a)}$, need to be calculated. Both can be obtained by using transposed convolutions:

$$\nabla l_{\mathbf{A_i}} = \text{tconv}(\mathbf{W}, \nabla l_{\mathbf{A_o}})$$

$$\nabla l_{\mathbf{W}} = \text{pconv}(\mathbf{A_i}, \nabla l_{\mathbf{A_o}})$$

where tconv indicates the transpose convolution operator, and pconv indicates the permute convolution operator. Both of them can be defined in terms of a normal convolution (ignoring stride, padding, grouping, etc).

$$\text{tconv}(\mathbf{W}, \nabla l_{\mathbf{A_o}}) \coloneqq \text{conv}({\mathbf{W_{180}}}^T, \nabla l_{\mathbf{A_o}})$$

$$\text{pconv}(\mathbf{A_i}, \nabla l_{\mathbf{A_o}}) \coloneqq \text{conv}({\mathbf{A_i}}^T, \nabla {l}^T_{\mathbf{A_o}})^T$$

where $_{180}$ indicates a rotation of 180 degree across the third and fourth dimensions (w, h), and ^{T} means transposing in the first and second dimensions (i, b).

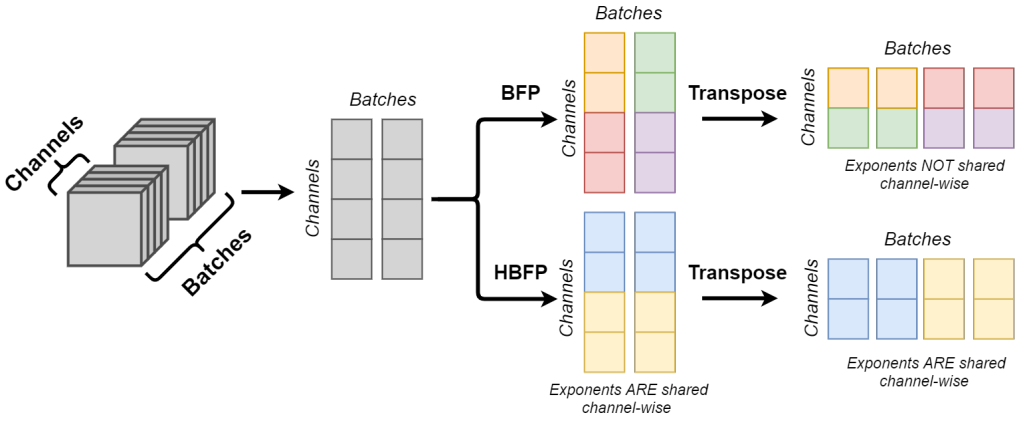


Figure 2: Difference between BFP and HyperBlock. In this example, the tensor has shape $(2, 4, 1, 1)$, such that batch$= 2$, channel$= 4$, width=height$= 1$, $blk = 2$. Same coloured cells correspond to the same shared exponent. In BFP, before transposition, cells were aligned such that Multiply and Accumulate (MAC) could be performed within blocks. After transposition, BFP does not hold this property, as numbers with shared exponent are in different batches. On the other hand, HBFP keeps shared exponents in the same batches after transposition, since they are shared across a 2D block of 2 by 2.

Note that for both tconv and pconv, the first two dimensions need to be transposed, which causes problems when applying BFP operations, since the exponent that was previously shared across depth is now shared across the batch / outer channel dimension, which means that MACs using integer only operations can no longer be exploited, since they do not share an exponent anymore. Figure 2 shows the problem for BFP. After transposition of channels and batches / output channels dimensions, exponents are not shared anymore, which makes it impossible to take advantage of low bitwidth MAC operations in hardware.

There are two ways to avoid this "permutation problem": Direct Gradient BFP, and HyperBlock Floating Point. We compare both in the next sections.

4.1. Direct Gradient Block Floating Point

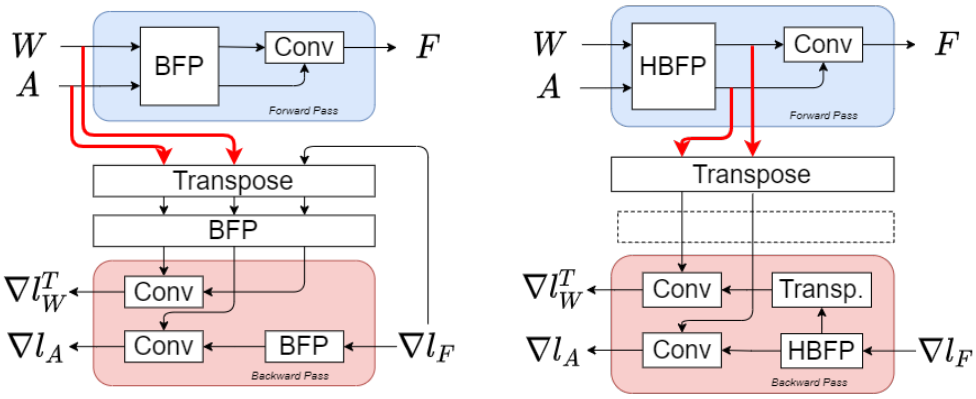
Naively applying BFP to gradients means adding a BFP quantizer just after transposition, but before convolution. This means that just before a tensor is convolved, BFP quantizer is applied across the dimension that is going to be performed the dot-product. In the forward pass, this means applying BFP depthwise on both the weight and activation. In the backward pass, this means applying BFP across batches in the activation and gradient in order to convolve them resulting in the gradient with respect to the weight; and across output channel in the weight and across channel in the gradient to perform the convolution, resulting in the gradient with respect to activation. Figure 3a shows how this is done.

Note that because BFP is not invariant to permutation, the quantization of the tensors in the normal and in the permuted dimension need to be applied every time they are convolved. This has two implications: First, in total there is a need to quantize tensors 6 times for each convolutional layer - two in the forward pass for the weight and activation, and 4 in the backward pass for the transposed version

Algorithm 1: Block Floating Point Quantizer

Input: Input[N, C, W, H], BIT, BLK
Result: IntT[N, C, W, H], ExpT[N, blk, W, H];

```
blk = ceil(C/BLK);
// Part 1: Assemble ExpT
for n=0:N; w=0:W; h=0:H; b=0:blk do
    max_e ← -127;
    for c_blk=0:BLK do
        c ← c_blk + b * BLK;
        e ← get_exp(Input[n, c, w, h]);
        if e > max_e then
            max_e ← e;
        end
    end
    ExpT[n, b, w, h] ← max_e;
end
// Part 2: Assemble IntT
for n=0:N; w=0:W; h=0:H;c=0:C; do
    e ← get_exp(Input[n, c, w, h]);
    m ← mantissa(Input[n, c, w, h]);
    s ← sign(Input[n, c, w, h]);
    shift ← (ExpT[n, floor(c/BLK), w, h] - e);
    m = (m >> shift) + U(0, 1);
    m = floor(m, BIT);
    IntT[n, c, w, h] = s && m;
end
```



(a) Convolution Layer using BFP

(b) Convolution Layer using HBFP

Figure 3: Diagram showing forward and backward passes of Direct Block Floating Point (DBFP) (a), and HyperBlock Floating Point (HBFP) (b). In DBFP, there are 6 BFP quantization operations, whereas just 3 are needed in HBFP since they are invariant to transposition. As highlighted by the red arrows, in DBFP the FP32 tensors are used twice, whereas in HBFP, the already quantized tensors are reused for backprop. Following previous papers [1, 45], only $\nabla l_{\mathbf{A}}$ is quantized, whereas $\nabla l_{\mathbf{W}}$ used for the update step is kept in FP32.

of weight, activation, and gradient with respect to feature map; Second, since different dimensions are being quantized on the fly, all tensors need to be stored in FP32, which consequently requires considerably higher bandwidth.

Algorithm 1 details this implementation. First, for each block in the tensor, the maximum exponent is extracted. For a block size of BLK, this results in an exponent tensor with sizes $(N, \text{ceil}(C/BLK), W, H)$. Therefore, there is approximately $NCWH$ multiplies in low-bit fixed point, and $NWH\text{ceil}(C/BLK)$ adds in FP32 for each tensor multiplication. Note that this can be parallelized in the outer loop. The same implementation works for transposition by just changing the index of the first and second dimensions. Once the exponent tensor is assembled, a shift operator can be applied to each value in the input tensor, based on which block they belong to, such that they all share the exponent dictated by the exponent tensor. Stochastic rounding [3] is then implemented (by adding a Uniformly sampled value as explained in section 4.3), and the final value is stored in the integer tensor, depending on how many bits is used. Note that this second part is parallelized in all four dimensions. This results in an Integer Tensor that performs all the multiplications, and the bulk of additions during any convolution.

4.2. HyperBlock

We propose a simpler way of avoiding having to quantize the same tensors multiple times. Instead of quantizing using blocks depthwise, 2 dimensional blocks can be used across the two dimensions that are transposed. Figure 2 shows how this is done. First, a block value is selected (say $blk = 2$). This means that for a 2 dimensional block of 2x2, an exponent is selected and shared across the entire 2D block, resulting in an exponent tensor with sizes $(\text{ceil}(N/BLK), \text{ceil}(C/BLK), W, H)$. Therefore, there are $NCWH$ fixed-point multiplies and $WH\text{ceil}(NC/BLK^2)$ adds in FP32 for each tensor multiplication. Whenever the tensor needs to transpose before convolutions, the right blocks still share the same exponents, and there is no need to quantize further.

This has many implications for the entire process, which can be seen in Figure 3b. For the forward pass, the weight and activation are quantized using HyperBFP and convolved. Note that for the backward pass, the already quantized version of the weight and the activations are used, which means that they do not need to be stored in FP32. For the backward pass, the now quantized weight and activation are transposed, together with the gradient with respect to the output map, and they are convolved as expected. This means that the number of quantization operations needed is 3 for each layer, and that once quantized, their floating point equivalent do not need to be stored anymore, considerably reducing the memory bandwidth requirement.

Algorithm 2 shows how the HyperBlock quantizer works. Similary to the BFP Quantizer, the Exponent Tensor is assembled by finding the maximum exponent across these 2 dimensional blocks. Then the exponents are use to shift each value of the input tensor, and then it is stochastically rounded to the desired bitwidth. All of these operations are massively paralellizeable, and since they only involve reading the mantissa and exponent values, and shifting, they can be efficiently implemented in custom hardware.

Algorithm 2: HyperBlock Quantizer

Input: Input[N, C, W, H], BIT, BLK
Result: IntT[N, C, W, H], ExpT[blkn, blkc, W, H];

```
blkc = ceil(C/BLK);
blkn = ceil(N/BLK);
// Part 1:  Assemble ExpT
for w=0:W; h=0:H; b_c=0:blkc; b_n=0:blkn do
    max_e ← -127;
    for c_blk=0:BLK; n_blk=0:BLK do
        c ← c_blk + b_c * BLK;
        n ← n_blk + b_n * BLK;
        e ← get_exp(Input[n, c, w, h]);
        if e > max_e then
            max_e ← e;
        end
    end
    ExpT[b_n, b_c, w, h] ← max_e;
end
// Part 2:  Assemble IntT
for n=0:N; w=0:W; h=0:H;c=0:C; do
    e ← get_exp(Input[n, c, w, h]);
    m ← mantissa(Input[n, c, w, h]);
    s ← sign(Input[n, c, w, h]);
    shift ← (ExpT[floor(n/BLK), floor(c/BLK), w,
      h] - e);
    m = (m >> shift) + U(0, 1);
    m = floor(m, BIT);
    IntT[n, c, w, h] = s && m;
end
```

4.3. Exponent Selection and Stochastic Rounding

In previous papers, the exponent selection is made by sharing the largest exponent across the entire block [42]. This guarantees that the important (large) values are unaltered. We have also adopted this strategy in this paper.

Note that stochastic rounding is used in both methods, which means rounding a value x up with probability $p = \frac{x-\lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor}$, and down with probability $1-p$. In practice, we simply sample from a uniform distribution, sum to the value, and floor the result $x = \lfloor x + U(0,1) \rfloor$. The proof of this equivalence is provided in the supplementary material.

5. Results

We compare the two different approaches for forward and backward pass calculations: We apply DirectBFP and HyperBlock method on Cifar10, SVHN and ImageNet.

CIFAR10 We tested our method on the CIFAR10 dataset [17], which consists of 60k images in total, each with 3 colour channels, in a 32x32 pixels format. The dataset is split between 10k images for testing and 50k images for training, each labelled with 10 different classes. We trained all networks for 200 epochs, using a Stochastic Gradient Descent optimiser, with initial learning rate of 0.01, momentum of 0.9, weight decay of 5e-4, and reducing the learning rate by a factor of 10 every 60 epochs.

ResNet18 HyperBlock

		BLOCK 8	BLOCK 16	BLOCK 32
BIT	4	94.4% (-0.1)	94.5% (-0.0)	94.3% (-0.2)
	3	94.1% (-0.4)	93.6% (-0.9)	92.7% (-1.8)
	2	93.0% (-1.5)	89.2% (-5.3)	78.0% (-16.5)

ResNet18 DirectBFP

		BLOCK 8	BLOCK 16	BLOCK 32
BIT	4	94.1% (-0.4)	93.4% (-1.1)	93.7% (-0.8)
	3	94.0% (-0.5)	93.5% (-1.0)	80.0% (-14.5)
	2	93.3% (-1.2)	69.5% (-25.0)	66.0% (-28.5)

Table 1: Results of the HyperBlock and Direct methods applied to ResNet18 using the CIFAR10 Dataset, which achieves an accuracy of 94.5% using FP32. The number in parenthesis indicate difference in accuracy to FP32 model.

The results are shown in Table 1 for ResNet18, and in Table 2 for the VGG16. In both cases the bit value shown represents the bit-length that the weights, activations and gradients of all but the first convolutional layer were quantized. In both tables we can see the influence that bitlength, block size, and method choice affects the results. As expected, the lower the bitlength, the harder it is to train the models and the lower is the accuracy. It is important to note that a lower bitlength corresponds to more efficient computation, so there is a natural trade-off between computational complexity and accuracy. The block size affects the accuracy as well. The lower the block size, the better the accuracy is, but less efficient is the computation. Note however that the block size affects differently depending on the method chosen. The results show that for extremely low bitwidths (2 and 3), the Direct method loses accuracy quicker than HyperBlock as the block size increases. For example, in Table 2, there was a loss of 27% when increasing the block size from 8 to 32 in the case of using 2 bits in the Direct method, whereas there was a loss of only 9% when using the HyperBlock method. The same happens in Table 1, where there was a loss of 15% when using HyperBlock, whereas there was a loss of 27% when the Direct method is used.

We believe this is because the weight tensor used for the forward pass is different than the weight tensor used

VGG16 HyperBlock

		BLOCK 8	16	32
BIT	4	93.4% (0.8)	93.2% (0.6)	93.4% (0.8)
	3	92.9% (0.3)	93.2% (0.6)	92.6% (-0.0)
	2	92.8% (0.2)	88.9% (-3.7)	83.3% (-9.3)

VGG16 DirectBFP

		BLOCK 8	16	32
BIT	4	92.9% (0.3)	92.9% (0.3)	92.5% (-0.1)
	3	92.5% (-0.1)	93.0% (0.4)	79.2% (-13.4)
	2	92.3% (-0.3)	76.2% (-16.4)	65.4% (-27.2)

Table 2: Results of the HyperBlock and Direct methods applied to VGG16 using the CIFAR10 Dataset, which achieves an accuracy of 92.6% using FP32. The number in parenthesis indicate difference in accuracy to FP32 model.

Model	W	A	G	Block	Method	Top-1
Model A_{dorefa}	2	2	2	-	-	90.0%
Model A_{ours}	**2**	**2**	**2**	**8**	**D**	**97.4%**
Model A_{ours}	2	2	2	16	D	85.7%
Model A_{ours}	4	4	4	16	H	96.9%
Model A_{ours}	4	4	4	32	H	95.8%
Model B_{dorefa}	2	2	2	-	-	91.9%
Model B_{ours}	**2**	**2**	**2**	**8**	**D**	**96.6%**
Model B_{ours}	2	2	2	16	D	86.7%
Model B_{ours}	4	4	4	16	H	96.6%
Model B_{ours}	4	4	4	32	H	96.0%
Model C_{dorefa}	2	2	2	-	-	85.6%
Model C_{ours}	**2**	**2**	**2**	**8**	**D**	**91.4%**
Model C_{ours}	2	2	2	16	D	75.7%
Model C_{ours}	4	4	4	16	H	96.3%
Model C_{ours}	4	4	4	32	H	94.4%

Table 3: Comparison between our method and the one of DoReFa-Net [49] on the SVHN dataset. The "Direct" in method indicates usage of the DirectBFP method, and the rows with "Hyper" indicates HyperBFP.

in the backward pass in the Direct method. Therefore, it is likely that the gradient computation would be impacted. In the Hyperblock method on the other hand, the same quantized weight used for the forward pass is also used in the backward pass, which means that the gradient computed is likely more representative of the right direction for the update step.

SVHN In order to compare with other methods, we also tested our algorithm on the SVHN dataset, which is similar to CIFAR10. This consists of approximately 73k digits for the regular training set, and 26k digits for the testing set. Just like DoReFa-Net [49], we also used an extra training set consisting of 531k digits. All images are labelled with one of the 10 digits, and each are cropped to 32x32 pixels with 3 channels for colours, and resized to 40x40 pixels.

We implemented DoReFa-Net's model architecture for fair comparison. Model A consists of 8 layers: seven convolutional layers with sizes 48, 64, 64, 128, 128, 128, 512, and an FC layer at the end. Models B, C and D are achieved by multiplying the channels by 0.5, 0.25, and 0.125 respectively. Table 3 shows the results. Although our aim is not to get extremely low bitwidth, we have competitive results even when using only 2 bits. For relatively low block sizes (value of 8), our network outperforms DoReFa-Net in many instances, when using DirectBFP method. We achieve within 1% of accuracy using the HyperBlock method using only 4 bits and block sizes of 16 to 32.

ImageNet We also trained our model on ImageNet [30]. We used the ILSVRC2012 training set, which consists of 1.2 million images, and the validation set, consisting of 50k images. They are all normalized and cropped to 224x224 pixels, with 3 colour channels before being fed to the network. We use batch of 128 and SGD with learning rate of 0.01, just as indicated in the AlexNet paper [18]. For the ResNet18 training, we used learning rate of 0.1 multiplying it by 0.1 every 30 epochs following the ResNet paper [9].

Based on the results from both SVHN and CIFAR10, we tested using block size of 8 when using bitwidth of 2, and between 16 to 32 for bitwidth of 4 or more. The results are shown in Table 4. There is no loss of accuracy when using only 4 bits for weights, activations and gradients using AlexNet, for block size of 16, and a small loss when using block size of 32. Both DirectBFP and HyperBFP achieved good results when using block size of 32 using only 4 bits. There is an accuracy loss of 8% when using only 2 bits with block size of 8. This is encouraging as even though our method is not meant to provide good accuracy in extreme low bitwidth, we are still competitive with DoReFa-Net.

ResNet18 is a more challenging network to train. We still get competitive results with 8 bits, achieving less than 1% accuracy loss for 8 and 5 bits. In comparison to SWALP, this is achieved without the need to introduce any extra logic to the hardware to perform averaging of weights. It also avoids the need for hyperbolic tangent computations needed in DoReFa-Net. The accuracy degrades rapidly when going to lower bits though, and lower block sizes need to be used. We also compared to ALQ/AMQ [7] and TRN [39], even though they solve a different problem than us, since they aim to quantize only the gradients. We include these results

Algorithm	W	A	G	Block	Method	Model	Top1
DoReFa [49]	1	2	6	-	-	AlexNet	46.1%
Ours	2	2	2	8	H	AlexNet	48.8%
Ours	**4**	**4**	**4**	**16**	**H**	**AlexNet**	**56.2%**
Ours	4	4	4	32	H	AlexNet	55.6%
Ours	4	4	4	32	D	AlexNet	55.4%
DoReFa [49]	8	8	8	-	-	AlexNet	53.0%
FP32	32			-	-	AlexNet	55.9%
Ours	**8**	**8**	**8**	**32**	**D**	**ResNet18**	**69.2%**
Ours	5	5	5	32	D	ResNet18	68.3%
ALQ [7]	32	32	3	-	-	ResNet18	67.7%
SWALP [42]	8	8	8	-	-	ResNet18	65.1%
AMQ [7]	32	32	3	-	-	ResNet18	64.8%
TRN [39]	32	32	2	-	-	ResNet18	62.8%
Ours	8	8	8	64	D	ResNet18	61.5%
Ours	4	4	4	32	D	ResNet18	57.1%
FP32	32			-	-	ResNet18	69.7%

Table 4: Results of running AlexNet and ResNet18 on ImageNet using different methods. We can achieve FP32 results using only 4 bits for weights, activations and gradients.

here since they provide an estimate of an "upper bound" on accuracy for quantized gradients. We can see that a significant drop in accuracy is expected when using low precision gradients regardless of the precision used in weights and activations, which illustrates the difficulty of this task.

Role of Stochastic Rounding We use stochastic rounding for both BFP and HBFP. Without stochastic rounding, our method's accuracy drops on 5-bit ResNet18 from 68.3% to 40.7%, a loss of 27.6%. We believe that this is so effective because it avoids the problem of *stagnation*, where gradients are rounded down to zero, which stagnates the weight values in suboptimal values. More details about our conclusion is provided in the supplementary material.

5.1. Hardware Comparisons

To show the advantages of being able to perform training and inference by using our method, we created (gate level VHDL [13]) a wide range of BFP operations (used in both of our methods) and floating point arithmetic operators, and synthesized these to a current production 10nm FinFET library with a target frequency of 800MHz. We also compared a subset of results to Synopsys DesignWare [33] components synthesized with the same libraries and scripts; both methods were always within 10% area. We report area in Table 5: the "Mantissa Multiplier" corresponds to the area cost of multiplying the mantissa portion of the formats; the "FP Multiplier" includes also the operations needed to include the exponent part of the floating point formats; the "FP ALU" is the cost of accumulating the result of the multiplications in the block; the "Cost" is the sum of the "FP Multiplier" and the "FP ALU". 'Relative Cost" is the normalized "Cost" to the smallest operation. To more accurately model the area of these operators, we built a system level construct for each, with pipeline registers before and after the floating point operators. Additionally, FP ALUs also had wrapped multiplexers to model the cost of the dynamic selection of addition and accumulation operations. The BFP functions were implemented as tensors of 10 element dot products, again bounded by registers [21]; the cost of the fixed to floating point conversions is included in the table, amortised over the tensor per multiplier.

	Mantissa Multiplier	FP Multiplier	FP ALU	Cost	Relative Cost
FP32	156[1]	186[1]	150 [1]	336	37
TF32 [29]	32 [1]	52[2]	90	142	16
Bfloat16 [38]	19 [1]	40 [2]	150	190	21
FP16 [12]	32 [1]	40 [2]	70	110	12
FP8 [37]	6 [1]	20 [2]	30[2]	50	5.6
INT8 BFP	15 [1]	15	10	25	2.8
INT4 BFP	4 [1]	4	5	9	1

[1] Run on Synopsys DC [2] Estimated

Table 5: Comparison of cost in different number formats.

From Table 5, it can be seen that the BFP formats are by far the smallest. For INT8, we use 10x less area than FP32, and we are also at least 2x more efficient than the common formats used for training, such as TF32 [29], BFloat16 [38], and FP16 [12]. In terms of relative cost, we are also 2x more efficient than the emerging FP8 format [37]. The difference is still larger when using INT4. Saving half of the area means that we can pack twice the number of operations, which corresponds to approximately twice the throughput.

The importance of area goes beyond saving silicon, power, and increased throughput. Memory accesses are very expensive [10], with 1 or 2 orders of magnitude for local embedded SRAM (based on analysis of cache memory at 45nm), and 3 or 4 orders of magnitude for DRAM access. HyperBFP has the potential of getting further benefits from memory accesses when comparing to DirectBFP for example, since the same quantized tensor can be used in both forward pass and backward passes, without the need to retrieve expensive FP32 values as is the case when using DirectBFP. The more processing we can perform in the arithmetic datapath, the more efficient the system will be.

6. Conclusion

We have demonstrated that DirectBFP and its HyperBlock variant can be used for quantizing gradients during training of convolutional neural networks. We have achieved virtually no loss of accuracy when training AlexNet on the challenging ImageNet dataset, by using multiplications in fixed-point, and a smaller number of addition using floating point format. We believe that this opens up space for efficient implementation in custom hardware of low bit training and inference.

References

[1] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *arXiv preprint arXiv:1805.11046*, 2018.

[2] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5918–5926, 2017.

[3] Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Théo Mary, and Mantas Mikaitis. Stochastic rounding: Implementation, error analysis, and applications. *Royal Society Open Science*, March 2022.

[4] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in Neural Information Processing Systems*, 33, 2020.

[5] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[6] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. *arXiv preprint arXiv:1804.01526*, 2018.

[7] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. Adaptive gradient quantization for data-parallel sgd. *Advances in neural information processing systems*, 33:3174–3185, 2020.

[8] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 1–14. IEEE Press, 2018.

[9] 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.

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

[11] 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.

[12] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[13] IEEE. Ieee standard vhdl language reference manual - redline. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) - Redline*, pages 1–620, 2009.

[14] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. AI benchmark: All about deep learning on smartphones in 2019. *CoRR*, abs/1910.06663, 2019.

[15] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[16] Jangho Kim, KiYoon Yoo, and Nojun Kwak. Position-based scaled gradient for model quantization and pruning. *Advances in Neural Information Processing Systems*, 33:20415–20426, 2020.

[17] Alex Krizhevsky et al. Learning multiple layers of features from tiny images, 2009.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[19] Habana Labs. Gaudi training platform white paper. Technical report, Habana Labs, November 2020. Accessed on 20/02/2022.

[20] Haban Labs. Goya inference platform white paper. Technical report, Habana Labs, November 2020. Accessed on 20/02/2022.

[21] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. Stratix 10 nx architecture and applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 57–67, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Junghyup Lee, Dohyung Kim, and Bumsub Ham. Network quantization with element-wise gradient scaling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6448–6457, 2021.

[23] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. Brecq: Pushing the limit of post-training quantization by block reconstruction. *arXiv preprint arXiv:2102.05426*, 2021.

[24] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In I. Guyon,

U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[25] Yuang Liu, Wei Zhang, and Jun Wang. Zero-shot adversarial quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1512–1521, 2021.

[26] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pages 7197–7206. PMLR, 2020.

[27] Marcelo Gennari do Nascimento, Roger Fawcett, and Victor Adrian Prisacariu. Dsconv: Efficient convolution operator. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[28] M. G. D. Nascimento, V. Prisacariu, and R. Fawcett. Dsconv: Efficient convolution operator. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5147–5156, 2019.

[29] NVIDIA. Training neural networks with tensor cores NVIDIA. https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf. Accessed: 2020-03-08.

[30] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[32] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[33] Synopsis. Synopsis designware. https://www.synopsys.com/simpleware.html. Accessed: 2022-22-08.

[34] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[35] 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.

[36] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

[37] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.

[38] Shibo Wang and Pankaj Kanware. Bfloat16: The secret to high performance on cloud tpus. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. Accessed: 2020-03-08.

[39] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems*, 30, 2017.

[40] 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/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[41] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[42] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Chris De Sa. Swalp: Stochastic weight averaging in low precision training. In *International Conference on Machine Learning*, pages 7015–7024. PMLR, 2019.

[43] Yukuan Yang, Lei Deng, Shuang Wu, Tianyi Yan, Yuan Xie, and Guoqi Li. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks*, 125:70–82, 2020.

[44] 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)*, September 2018.

[45] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, et al. Fixed-point back-propagation training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2330–2338, 2020.

[46] Xiangguo Zhang, Haotong Qin, Yifu Ding, Ruihao Gong, Qinghua Yan, Renshuai Tao, Yuhang Li, Fengwei Yu, and Xianglong Liu. Diversifying sample generation for accurate data-free quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15658–15667, 2021.

[47] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[48] Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. Distribution adaptive int8 quantization for training cnns. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021.

[49] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Supplementary Material - HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation

Marcelo Gennari do Nascimento
University of Oxford - Active Vision Lab
Parks Road, Oxford OX1 3PJ
marcelo@robots.ox.ac.uk

Victor Adrian Prisacariu
University of Oxford - Active Vision Lab
Parks Road, Oxford OX1 3PJ
victor@robots.ox.ac.uk

Roger Fawcett
Intel Corporation
Eclipse, Globe Park, Marlow SL7 1YL
roger.fawcett@intel.com

Martin Langhammer
Intel Corporation
Eclipse, Globe Park, Marlow SL7 1YL
martin.langhammer@intel.com

1. Stochastic Rounding Equivalence Proof

Stochastic rounding is an umbrella term that encompasses many different methods. The most common method is setting the probability of rounding up to be proportional to the normalized distance between the value and its ceiling. Mathematically, let x be a real number, $\lceil x \rceil$ be its rounded-up (ceiling) value, and $\lfloor x \rfloor$ be its rounded-down (floor) value. The most common stochastic rounding procedure used in deep learning consists of the following function:

$$f(x) = \begin{cases} \lceil x \rceil & w.p. \quad p = \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor} \\ \lfloor x \rfloor & w.p. \quad p = \frac{\lceil x \rceil - x}{\lceil x \rceil - \lfloor x \rfloor} \end{cases} \tag{1}$$

assuming for example that $x \in [0, 1]$, then if $x = 0.8$, the probability that it will round to 1 is 80%. In hardware a stright-forward way of implementing this operation is by simply adding a number sampled from a uniform distribution, and flooring its value:

$$\begin{aligned} \nu &\sim U(0, \lceil x \rceil - \lfloor x \rfloor) \\ f(x) &= \lfloor x + \nu \rfloor \end{aligned} \tag{2}$$

where $U(0, a)$ is a uniform distribution from 0 to a. This has a close relation to the idea of *dithering* [2, 1], in which noise is intentionally added to the signal in order to suppress error. In the following part, we proof this equivalence.

Theorem 1.1. *The function $f(x) = \lfloor x + \nu \rfloor$ for $\nu \sim U(0, \lceil x \rceil - \lfloor x \rfloor)$ is equivalent to stochastic rounding as defined in equation 1.*

Proof. The function $f(x)$ can be rewritten in the following format:

$$f(x) = \begin{cases} \lceil x \rceil & w.p. \quad p(x + \nu \geq \lceil x \rceil) \\ \lfloor x \rfloor & w.p. \quad p(x + \nu < \lceil x \rceil) \end{cases}$$

Note that for any value of x and any quantization gap $\lceil x \rceil - \lfloor x \rfloor$, we have that $\lfloor x \rfloor \leq x \leq \lceil x \rceil$.

$$p(x + \nu \geq \lceil x \rceil) = \int_{\lceil x \rceil}^{\infty} x + U(0, \lceil x \rceil - \lfloor x \rfloor) dy \tag{3}$$

$$= \int_{\lceil x \rceil}^{\infty} U(x, x + \lceil x \rceil - \lfloor x \rfloor) dy \tag{4}$$

$$= \int_{\lceil x \rceil}^{\infty} U(x, x + \lceil x \rceil - \lfloor x \rfloor) dy \tag{5}$$

$$= \int_{\lceil x \rceil}^{\infty} \frac{1}{x + \lceil x \rceil - \lfloor x \rfloor - x} \mathbb{1}_{y \in (x, x + \lceil x \rceil - \lfloor x \rfloor)} dy \tag{6}$$

where $\mathbb{1}_{y \in (a,b)}$ is the indicator function, which is equal to 1 when the condition $y \in (a, b)$ is satisfied, otherwise it is equal to 0.

By definition we have that $x < \lceil x \rceil$, therefore:

$$p(x + \nu \geq \lceil x \rceil) = \int_{\lceil x \rceil}^{x + \lceil x \rceil - \lfloor x \rfloor} \frac{1}{\lceil x \rceil - \lfloor x \rfloor} dy \tag{7}$$

$$= \frac{x + \lceil x \rceil - \lfloor x \rfloor - \lceil x \rceil}{\lceil x \rceil - \lfloor x \rfloor} \tag{8}$$

$$= \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor} \tag{9}$$

The proof for $p(x + \nu < \lceil x \rceil)$ is analogous. □

Theorem 1.2. *The function $f(x) = \lfloor x + \nu \rfloor$ for $\nu \sim U(0, \lceil x \rceil - \lfloor x \rfloor)$ is unbiased in expectation*

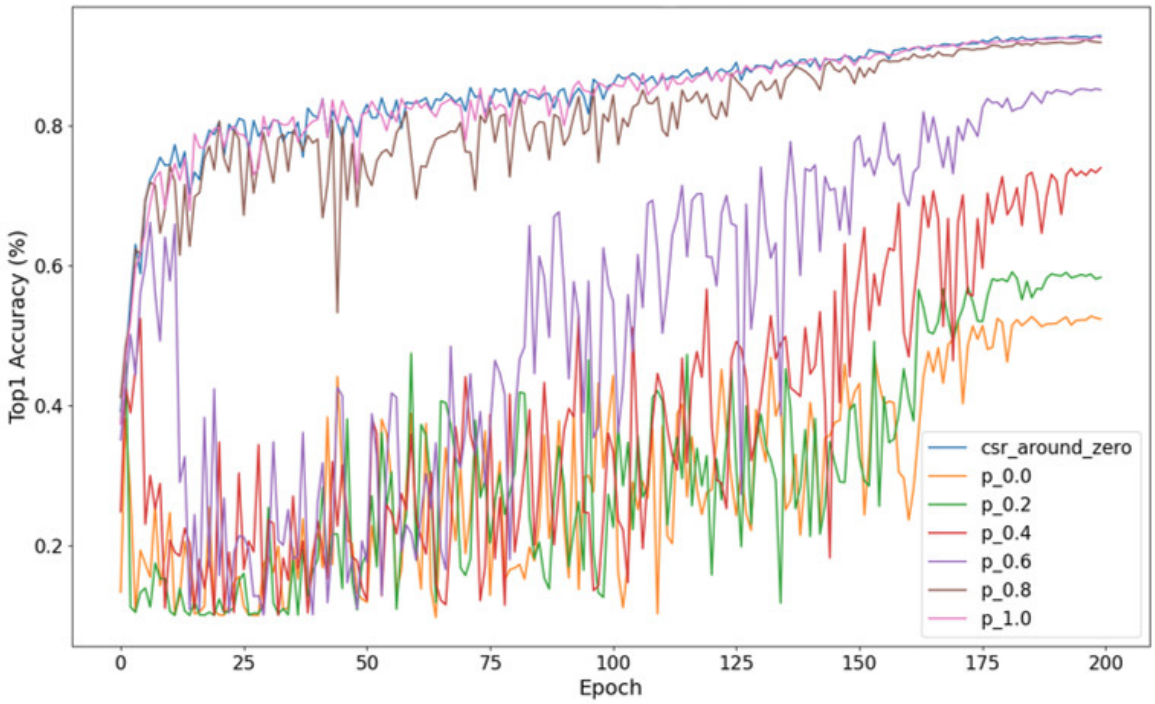


Figure 1. Stochastic Rounding using ResNet18 on CIFAR10 being applied at different percentages of tensor values, and around zero values.

Proof.

$$E[f(x)] = \lceil x \rceil \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor} + \lfloor x \rfloor \frac{\lceil x \rceil - x}{\lceil x \rceil - \lfloor x \rfloor} \tag{10}$$

$$= \frac{\lceil x \rceil x - \lceil x \rceil \lfloor x \rfloor + \lfloor x \rfloor \lceil x \rceil - \lfloor x \rfloor x}{\lceil x \rceil - \lfloor x \rfloor} \tag{11}$$

$$= x \tag{12}$$

□

2. Stagnation Problem Analysis

There are two main ideas for why to use Stochastic Rounding when doing quantization for training: i) it is an *unbiased* estimator of the real value of x; ii) it avoids *stagnation*, which happens when gradients get rounded to zero and therefore do not contribute to the learning of the neural network.

We have briefly investigated the reasons why stochastic rounding works and we have assembled the results in Figure 1. The figure shows the accuracy when training ResNet18 on CIFAR10 for different configurations of stochastic quantization. The lines labelled "Prob 0.x" means that at any iteration, only a random set of $x\%$ of the tensor values are being stochastically rounded, and the rest are rounding to the nearest value. For example, "Prob 0.2" means that a random 20% of the values are being stochastically rounded. The line labelled "csr_around_zero" means that only the values that would be rounded to zero are stochastically rounded.

The results show that using stochastic rounding on values around zero, which means avoiding *stagnation*, performs as well as using stochastic rounding in all of the values ("Prob 1.0"). When a low percentage of values are stochastically rounded, such as 20%, the training becomes too unstable and achieves significantly lower final accuracy. This indicates that the most likely reason why stochastic rounding works is because it avoids the weights of the neural network from getting stagnated at a certain value due to the gradients being rounded to zero. We believe this provides some insight into the reasons why stochastic rounding seems to be so effective when quantizing gradients specifically.

References

[1] L. Roberts. Picture coding using pseudo-random noise. *IRE Transactions on Information Theory*, 8(2):145–154, 1962.

[2] L. Schuchman. Dither signals and their effect on quantization noise. *IEEE Transactions on Communication Technology*, 12(4):162–165, 1964.

Statement of Authorship for joint/multi-authored papers for PGR thesis

To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation
Publication Status	□ Published □ Accepted for Publication X Submitted for Publication □ Unpublished and unsubmitted work written in a manuscript style
Publication Details	Nascimento, M. G. D., Fawcett, R., & Prisacariu, V. A., Langhammer M. (2022). HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation. *In Review for the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) 2023.*

Student Confirmation

Student Name:	Marcelo Gennari do Nascimento		
Contribution to the Paper	• Conception of all the ideas • Designed, and wrote the code for all software experiments • Performed all the experiments, apart from section "5.1 Hardware Comparisons" • Created all of the tables and diagrams, apart from "Table 5" • Wrote the manuscript, apart from section "5.1 Hardware Comparisons"		
		Date	12/09/2022

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Victor Adrian Prisacariu			
Supervisor comments The description on contributions is accurate			
Signature		Date	12/09/2022

This completed form should be included in the thesis, at the end of the relevant chapter.

5.5 Conclusion

In this chapter we introduced *HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation*, which is a method to quantize gradients and allow for both forward and backward passes to be performed using integer operations for the bulk of the computation. This is important because the backward pass require 2x more operations than the forward pass.

We achieve this by extending the block floating point format introduced in Chapter 3. Since the exponent in DSConv was shared channel-wise, it was not immediately transferable to the backward pass because of the tensor transposition needed for gradient calculation. Instead, we proposed to share the exponent in two dimensions, input-channel and output-channel (or batch-wise for activations), which makes the tensors transposition invariant.

We tested our method in the main neural network architectures using the SVHN, CIFAR10, and ImageNet dataset. We were able to train a ResNet18 using 8 bits with negligible accuracy degradation, and using 5 bits with less than 2% drop in accuracy.

6

A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks

Contents

6.1 Context 85
6.2 The usage of Neural Network Accelerators 85
6.3 Author Contribution 89
6.4 Publication: Gennari do Nascimento *et. al. "A Knowledge Distillation Approach toPruning and Accelerating SuperresolutionNeural Networks" In Prep.* 89
6.5 Conclusion 110

In this Chapter, we will mainly present our work "A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Network", which was written during my internship at Microsoft. We first start by providing some context on the problem formulation and the motivation of this work. We will then provide some background information on current neural network accelerators and how to exploit their benefits. We will then introduce our paper, and conclude with an overview of what has been achieved.

6.1 Context

A significant limitation of the quantization methods presented in the last chapters is that most commodity hardware do not support them. For real world applications, the methods above would have to be implemented either in reprogrammable devices such as *FPGA*s (Field Programmable Gate Arrays), which have a high-learning curve and labour costs to develop, or custom hardware devices such as *ASIC*s (Application Specific Integrated Circuits), which have high production and development costs. The alternative is to leverage the support that is currently available in hardware. This chapter will focus on briefly exploring the considerations for neural network topologies, number of multiply accumulates, and support for commercial hardware.

The goal of this chapter is to find a pipeline to compress a neural network architecture to run on the NPU provided on the Snapdragon 888 [32]. The task chosen (superresolution) is both due to commercial interest, and the fact that superresolution networks pose additional challenges to compression, for example they are notably more unstable to train as it is an ill-posed problem (and also due to the common use of Generative Adversarial setups), and the fact that the networks contain unorthodox layers, such as the Pixel Shuffle layer [141], which is a memory-intensive operation rather than compute-intensive.

6.2 The usage of Neural Network Accelerators

Because of the highly flexible and useful properties of neural networks, the industry has developed many different chips to accelerate convolutional neural networks. The internals of these chips are often black boxes due to patents and trade secrets, therefore it is hard to understand exactly where they can provide the most benefits. Furthermore, different chips have different characteristics, and perform better under different circumstances. It is therefore challenging to find ways of optimizing neural networks for specific target devices. A possible approach is to use a NAS algorithm, however this comes at a great cost on both time and energy. The goal in this section is to create an algorithm that can use minimal amount of computational

resources by compressing an already trained neural network for a particular task, whilst also targeting a specific device and runtime environment.

Implications on FLOPs statistics One of the most used metrics to estimate performance on hardware is the total number of Floating Point Operations (FLOPs) or MAC (Multiply and Accumulate) operations needed to run a neural network. This metric has been used in a variety of papers for multiple purposes.

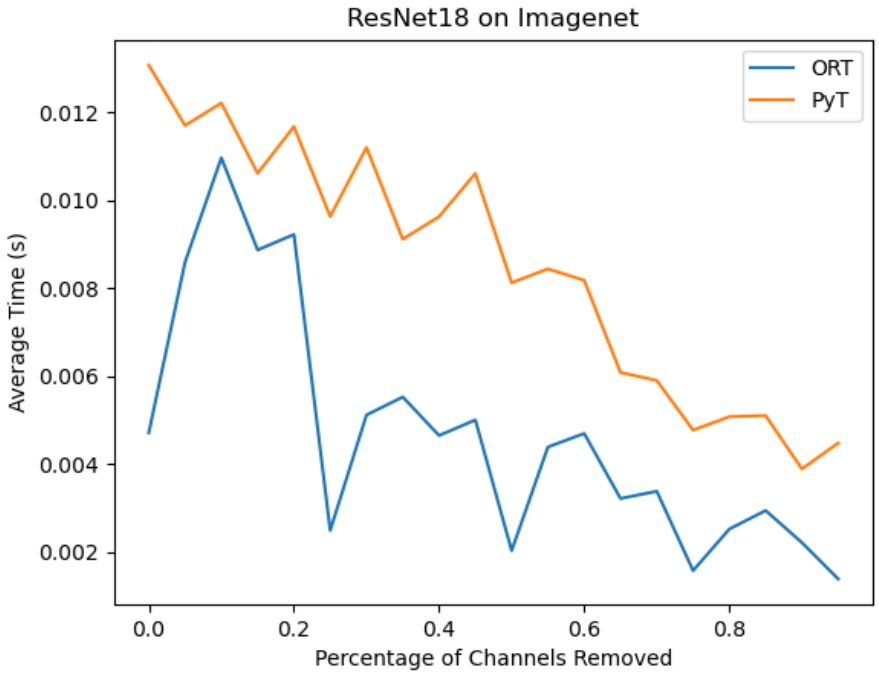
The argument to use this statistics is that it is hardware and software independent: different research groups use different libraries, compilers, and hardware to run neural networks, therefore timing results are mostly not reproducible. FLOPs statistics make the method independent from hardware and focuses only on a metric that is dependent only on the topology of the network chosen.

However, there are many limitations with this approach: i) FLOPs do not account for memory manipulation; ii) FLOPs do not account for parallelism; iii) FLOPs do not account for accelerators that target only specific layers; iv) FLOPs do not account for compilation / software optimizations.

Figure 6.1 shows an example of this limitation. The x-axis indicate the percentage of channels from each layer that has been removed from the CNN, and the y-channel shows the average time (in seconds) that it takes to run one image in the CPU. We used two different software packages to measure the time: PyTorch [142], labelled "PyT", and ONNX Runtime [143], labelled "ORT". Note that the FLOPs value should decrease linearly with the number of channels removed. However, there are instances where an architecture with higher FLOPs achieve significantly better average inference time. These are more pronounced in the line labelled "ORT", in which divisions by two (0.0, 0.25, 0.5, and 0.75) achieved significantly better results than the rest.

For example, not removing any channels takes approximately 0.005s running on ORT, but when 10% of the channels are removed, this number jumps to 0.011s, more than twice the time, even though the number of FLOPs was reduced by 10%.

Figure 6.1: Time taken to run ResNet18 on ImageNet.



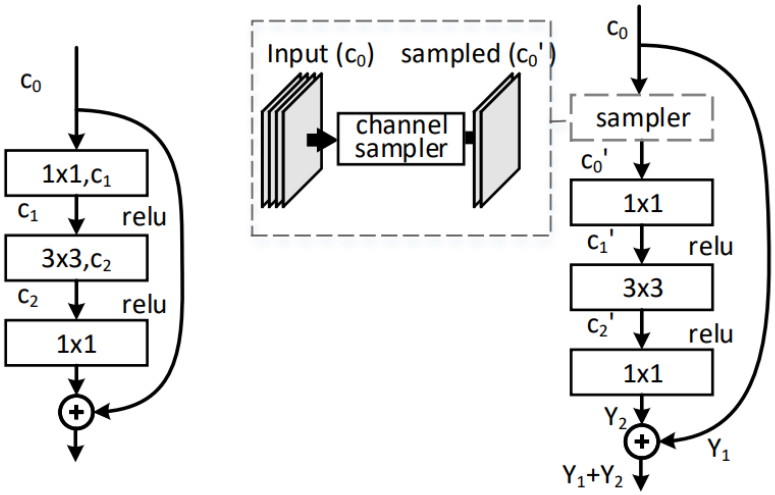
In this case the likely cause of this discrepancy is that ORT takes advantage of the optimized BLAS routines, which likely optimizes tensors that have number of channels equal to a power of 2. This is important because in practice, removing 25% of the channels will achieve better time and accuracy results than removing 70% of the channels.

Implications on memory usage Channel pruning is a common technique to use in order to compress a network for speed, that is because as explained in Chapter 2, it is a structured pruning technique that preserves dense tensors. One of the most popular ways of performing channel pruning is by selecting channels that are important in the architecture and removing the rest. Even though this has been shown to produce good accuracy results and smaller MAC requirements, often these methods neglect the amount of memory transport that needs to be used.

Figure 6.2 (taken from [104]) illustrates the problem. The network on the left requires a higher amount of FLOPs than the network on the right, since the channel sampler decreases the number of channels of the activation tensor from c_0 to c_0'. If

FLOPs is used as proxy of computational complexity, it would be expected that the network on the right performs quicker. However, the network on the right is also performing a considerable amount of memory manipulation, which is often not captured in the statistics when analysing the performance of neural networks.

Figure 6.2: Channel Sampler for branch problem. Image taken from [104].



Memory accesses still consume a significant amount of power, specially when tensors need to be fetched from outside of cache memory. This is illustrated in Figure 6.3. This table was taken from [144], and it shows the energy expenditure for a 45nm CMOS process. As it can be seen, SRAM operation required almost twice the energy of a 32bit floating point multiplication, and 37× the energy of a 32bit int addition. A DRAM read fares even worse: it required more than 100× the SRAM operation.

Figure 6.3: Energy consumption for common arithmetic and memory operations. Image taken from [144].

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit 32KB SRAM	5	50
32 bit DRAM	**640**	**6400**

This has also sparked a considerable effort from companies to design neural networks that can stay close to memory to maximize accesses from L1/L2 memory [145].

A fact that was easily noticed is that the NPU is not optimized for memory manipulation: it runs a convolutional layer quicker than a gather operation for example. This inspired the search for ways of performing channel pruning in branched topologies without the necessity of including channel selection layers, which involve a significant amount of memory manipulation.

Also following the observation of the iterative pruning approach, it was clear that performing the loop compression-retraining-compression would yield better results than using multiple compression stages and retraining only at the end. The culmination of these observations is a pipeline of many compression techniques that is bridged by knowledge distillation using the original network.

6.3 Author Contribution

- A method to accelerate CNNs on target devices by leveraging tensor decomposition, channel pruning, and quantization
- A pipeline for bridging compression techniques using knowledge distillation which allows multiple compression steps to be performed without losing significant accuracy
- Extensive results on target device with support for 8-bit quantization without the need to rely on proxy FLOPs metric

6.4 Publication: Gennari do Nascimento *et. al.* *"A Knowledge Distillation Approach toPruning and Accelerating SuperresolutionNeural Networks" In Prep.*

A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks

Marcelo Gennari do Nascimento[1,2][0000−0003−2025−5601], Karthik Vijayan[2], Sunando Sengupta[2], Alexandros Neopythou[2], and Eric Sommerlade[2]

[1] Active Vision Group, University of Oxford
[2] Microsoft Corporation

Abstract. Convolutional Neural Networks are a popular choice for Single Image Super Resolution (SISR) approaches. Although their representative power makes them the most suitable model, their computational load hinders its widespread use in commodity hardware. We propose a novel method to decompose, prune, and recover the degradation of superresolution networks via knowledge distillation for acceleration in commodity hardware. Previous methods often distill the knowledge of a teacher network with different depth to the student network. We use channel pruning to minimize the architectural gap between the student and the teacher, and thus achieve high FLOP (Floating Point Operations) reduction without significant image degradation. Our results show a reduction of FLOPs of 6.3x on SRGAN with less than 0.75dB degradation in PSNR. We also test our model on commodity hardware and show a speedup of 5.6x on SRGAN and 7.2x on EDSR.

1 Introduction

Convolutional Neural Networks have shown to have significant representative power and have been successfully employed in a wide range of computer vision tasks, including classification [27], image generation [12], image segmentation [11], and image superresolution [8]. With high definition monitors, mobile phones, and cameras widely available to the public, there is a rising need to either transmit high resolution images over the internet, or increase the quality of images taken from a low-quality camera. An interesting solution to this problem is using convolutional neural networks to upscale a low resolution image on the edge, which allows low resolution images to be captured, transmitted, and displayed on the edge as high resolution.

The problem with such approaches is that the representation power of neural networks comes at a high computational cost: older networks such as AlexNet [27] require 0.7G FLOPs, ResNet50 [16] require 4.1 GFLOPs, and even newer and efficient networks such as ShuffleNet [52] require 140 MFLOPs to classify a single 224x224 pixels image [5]. The number of GFLOPs required by superresolution networks are even higher than those required by classification networks: for instance, when running on Low Resolution (LR) images of size 125x125 pixels, with

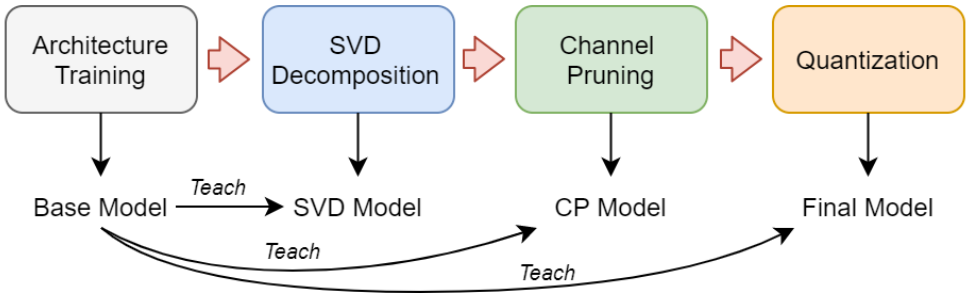


Fig. 1: Overview of the 4 phases in our method. We first train a base model, and use it as a teacher network to all models derived from it. Then we first perform spatial SVD model to transform each weight into a spatially separable tensor. Then we perform channel pruning on the weights to select which channels are important for final accuracy. Finally, in order to get the best out of the hardware available, we quantize the model to 8 bits. After each of these intermediate states, we perform knowledge distillation between the base model and the corresponding derived model.

an upsampling factor of 4x in each dimension, SRCNN [8] needs 26 GFLOPs, and RCAN [53] requires 359 GFLOPs [32]. When these models are run on commodity CPUs, SRCNN takes 1.8s and RCAN takes 51s [32], making it infeasible for these networks to run on commodity devices, such as mobile phones.

Taking advantage of the latest research on Neural Network compression, we propose a method to automatically decrease the number of total FLOPs required for superresolution tasks. Previous methods either rely on manual feature engineering or iterative search [9,37,47], or on a subset of neural network compression methods [18,20,31,39,43,50]. To the best of our knowledge, the majority of methods that employ knowledge distillation focus on distilling the knowledge from a deeper teacher to a shallower student network. Training a shallower student network via knowledge distillation can be tricky due to the well-known problem of the capacity gap between the teacher and the student [41]. If they are too structurally different, the student will learn less from the teacher.

Our method consists of four separate phases, as shown in Figure 1. 1) We train a base network using one of the common architectures found in the super-resolution literature; 2) To avoid the *branching problem* introduced in Section 3.2, we use Singular Value Decomposition (SVD) on each convolutional layer in order to split the convolution into two smaller convolutions, and we bridge the PSNR gap between this network and the base network by using Knowledge Distillation (KD) 3) We perform Channel Pruning on each convolutional layer, and we fine-tune as before by using KD; 4) We quantize the network to 8 bits to be run on a commodity hardware, and we again use KD to bridge the PSNR gap. Performing SVD decomposition and channel pruning enforces the student and the teacher networks to have similar structure and same depth. By

using this hierarchical approach to compression, we are capable of achieving a speedup of 5.6x when compared to the base network without significant losses in PSNR/SSIM or perception metrics.

2 Literature Review

High computational requirements is a problem that hinders the development of neural networks for many different tasks. In this section, we provide a brief overview of the important aspects in the literature that were used to help develop our method.

SuperResolution Since the introduction of SRCNN [8], convolutional neural networks have been extensively used for superresolution [23, 30, 49, 53, 54]. This is achieved by inputting a Low Resolution (LR) image with dimensions h and w and outputting a High Resolution (HR) image with dimensions nh and nw, where n is the upscaling factor. The architecture is usually trained using the L_2-norm. Ledig *et al* [30] noted that higher PSNR does not necessarily correspond to perceptually better looking pictures, often leading to overly smooth surfaces. They proposed SRGAN [30], which uses features from a VGG network to calculate an L_2 loss (content loss), and an adversarial loss by training a discriminator to differentiate between real and fake images. Since then, both methods have been extensively explored, with GAN-inspired networks including ESRGAN [49], and architectures that use solely the L-norm as a loss function including RCAN [53] and RDN [54].

Tensor Decomposition Due to the high computational requirements of neural networks, researchers have been looking at ways of minimizing the amount of MAC required to run each layer. One of the most promising methods is using Tensor Decomposition. This has been applied to learning separable filter banks in [44], which takes advantage of spatial redundancy to transform each filter $\mathbf{f_i^j} \in \mathbb{R}^{dxd}$ to two 1D filters $\mathbf{f_{i1}^j} \in \mathbb{R}^{dx1}, \mathbf{f_{i2}^j} \in \mathbb{R}^{1xd}$. One way of achieving this decomposition is by considering SVD in each filter and selecting only the first singular value. This reduces the number of multiplications from $\mathcal{O}(pqd^2)$ to $\mathcal{O}(2pqd)$, given input patch $x \in \mathbb{R}^{pxq}$. This technique has then been expanded in [7,24], by considering redundancies not only in the spatial dimension, but also in the channel dimension. Given a weight $W \in \mathbb{R}^{cxdxd}$, it can be decomposed into three 1D filters

Channel Pruning Following the seminal work "Optimal Brain Damage", LeCun *et al* [29] and the realisation that once trained, a network can be compressed by removing unnecessary weights, many recent approaches tried to take advantage of this fact to save memory and compute power. There are two main ways of pruning a network [3]. Firstly, *sparsity inducing* algorithms require the weights to be represented in a sparse format. This can be due to unstructured sparsity, such as intra-kernel pruning [14, 15], and even structured sparsity [3, 28], such

as kernel pruning. Even though very successful, this method makes it harder to take advantage of the hardware for fast compute because it generally require hardware specific implementation. Secondly, *non-sparsity inducing* algorithms do not require the weights to be represented in a sparse format. This is the case because the zeroed values can be completely removed from the tensor, transforming it in a shallower dense tensor. This is the case for example when performing layer-pruning, or channel-pruning [17, 36, 38]. These techniques are easier to take advantage of in hardware since they don't require any specialized implementation. For this reason, our approach focuses on non-sparsity inducing techniques.

Quantization Whereas the above methods aim to reduce the number of total MAC needed to run a network, quantization allows us to change from using floating point multiplication to using integer multiplications. Quantization consists of truncating weights and activations to lower bitwidth in order to use less computational resources per multiplication. Recent studies have demonstrated successfully that neural networks can be quantized, to 8-bits [48], ternary weights [33], and even 1-bit [22, 42]. With the widespread success of quantization techniques, the industry has developed hardware that is specially designed to run some form of quantized network. Since most commodity hardware accepts 8-bit quantization, this is the precision that we will focus on this work.

Knowledge Distillation A common way of training small models, is to use the knowledge of a bigger model to teach to the smaller one. This technique was introduced first by Buciluǎ, Caruana and Niculescu-Mizil [6], when they applied neural networks to mimic the behaviour of a model ensemble. This was followed by Hinton, Vinyals and Dean [19], where they coined they popularized the use of knowledge distillation, and introduced the idea of using the logits of a network, modified by a temperature parameter, as labels to another network. Since then, Knowledge Distillation has been adapted to be used to many situations. The most important adaptation for superresolution is the concept of *feature-based knowledge distillation* [13, 45], which consists of creating a loss function between intermediate feature maps from the teacher to the student. This is particularly helpful as it does not depend on using labels. Some papers that have used this technique isolated applied to superresolution [10, 18], but differently than us, they use it to teach a shallower network by using a deeper network. In this work, we tie together all of these different compression techniques using Knowledge Distillation as an intermediate step in order to keep model performance as close to the original as possible.

3 Method

Given a trained SR network, our method is composed of the four steps shown in Figure 1. The first step consists of the user providing a pretrained superresolution network. This will serve as the baseline and the teacher network. The second step

is to perform a spatial SVD decomposition in every convolutional layer, which results in transforming every weight kernel (k, k) into two: one with spatial size $(k, 1)$ and the other with size $(1, k)$. This process will be detailed in Section 3.1. The third step, detailed in Section 3.2 is to use Channel Pruning to remove unnecessary channels in the weights and activations. The last step detailed in section 3.3 is 8-bit quantization for fast inference, in order to take full advantage of the available commodity hardware that support quantized convolutions. In between all of these steps, we perform knowledge distillation between the user-provided network and its compressed version, with the algorithm exaplined in Section 3.4.

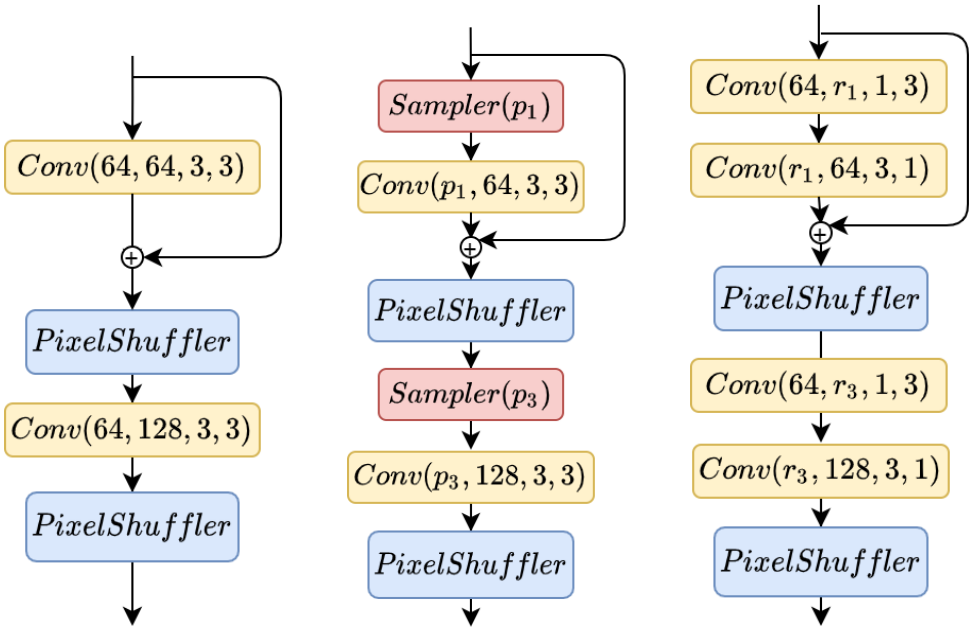
3.1 SVD Decomposition

The first step in our method is performing a spatial SVD Decomposition operation in every convolutional layer of our network. Given a kernel with size (I, O, K, K), where I is the number of input channels, O is the number of output channels, and K is the kernel size, we decompose this into two convolutional layers with size $(I, R, K, 1)$ and $(R, O, 1, K)$, where R is the desired rank of the tensor. The lower is the R value, the higher is the compression achieved. The SVD decomposition serves for two purposes: first, it reduces the MACs by guaranteeing that the weight tensor is separable and by choosing an appropriate rank value; second, it allows us to perform channel pruning in situations in which it would be impossible to do otherwise because of the topology of the network (this will be explained in Section 3.2).

3.2 Channel Pruning

After SVD decomposition, the next step is channel pruning. A common problem with channel pruning is that it cannot be performed in any topology, as shown in Figure 2. There are two situations in superresolution networks in which channel pruning cannot be performed: the first case is when there is a branch in the computational graph. This is caused because when performing channel pruning in a layer, in order for the feature maps to have consistent shape, the corresponding output channels in the previous layer also need to be removed. The problem is when there is a branch in the network and multiple layers need to remove channels that were not selected for pruning. In superresolution networks, a similar problem happens in convolutional layers that are before pixel shuffler layers. In the upsampling blocks, often Pixel Shuffling [46] is used to transform the input from (B, I, W, H) to $(B, I/n^2, nW, nH)$, where B is the batch size, I is the input channel, and n is an upscaling factor, generally equals to 2. In this case, the convolutional layer immediately before the pixel shuffler cannot be pruned naively without altering the final resolution of the image.

Many papers have created workarounds this problem. Most notably [17] proposes adding a channel sampler layer to guarantee that feature maps match the weight sizes. This method is applicable to both problems mentioned above, but it involves introducing a channel sampler layer in the network, which can be



(a) (b) (c)

Fig. 2: (a) Original Network, involving a residual and an upsampling layer. Channel Pruning is challenging in this topology because the branching and Pixel Shuffler [46] require the input and output sizes to be fixed. (b) The solution proposed by [17, 35], which involves including channel sampling layers, allows the convolutions to be channel pruned without affecting the adjacent layers (c) Our proposed solution, using spatial SVD decomposition and channel pruning. The rank r_n of each layer is determined by the channel pruning selection.

slow due to unstructured memory accesses. Alternatively, [35,38] guarantees the same input and output of each branched block, using channel pruning only on the middle layers. This method only applies to branched topologies, and would not work in reducing the MACs of the upsample part of the superresolution network.

Our solution is to use the SVD decomposition to decompose every layer into two sequential layers. This can solve both problems: When tackling branched topologies, we guarantee that the input and output of each branched block is the same by not performing any channel pruning. When tackling convolutions that are before a pixel shuffler, we now have decomposed our convolution into two, which makes it easy to perform channel pruning whilst guaranteeing the same input and output sizes.

3.3 Quantization

In the final phase of our model optimization method, we perform quantization in the compressed network. It is shown that full precision float operations are redundant and similar accuracy can be achieved by using half precision, or bfloat16 formats [25]. However, most of the commodity hardware have their best performance when the artihmetic operations are performed in integer precision, preferably in 8-bits. Reducing the precision from single precision to 8-bits usually results in a loss of accuracy, which has to be recovered by further training or fine-tuning. Since our method is aimed at running on commodity hardware, we focus on quantizing to 8-bits using static quantization, and then fine-tuning our model using knowledge distillation.

3.4 Knowledge Distillation

After performing the SVD decomposition, channel pruning, and quantization, the resulting intermediate networks will invariable have some deterioration in image quality. Trying to recover this deterioration by only retraining is not enough, due to the fact that the compressed network is smaller, which makes it harder to train. We propose to use Knowldge Distillation to help the compressed network to finetune by using the original uncompressed network as the teacher.

As a consequence of the work by Mizardeh *et al* [41], it is now known that it is harder for the student network to learn from the teacher if there is a large gap between their architectures. They showed that given teacher networks of varying depth, the student network might start to perform worse with deeper teacher networks. Inspired by this insight, we made two important design choices: first, we use knowledge distillation between networks with the same structure and depth. This means that the features used for the distillation should have the same level of abstraction in both the student and teacher networks. This is enforced by using only a subgraph of the teacher network by compressing it, instead of initializing a network of the same family but shallower as the student network (*e.g* using ResNet50 as the teacher and ResNet34 as the student). Secondly, we perform knowledge distillation after each of the compression steps, instead of performing it at the end only. Even though this takes longer to train, it helps ensuring that the student and teacher network never have a large difference in the output, which is crucial for the student network to recover as much from the compression step as possible.

The knowledge distillation algorithm is implemented by following the method introduced by [18]. The features extracted from the networks are the ones immediately following the residual groups in SRGAN. The loss function also follows [18]: we use L_2-loss for the features, and supervision from both the high-resolution image and the superresolved image from the teacher network.

4 Results

This section presents the results of our method. We also present the configuration needed to replicate our experiments. In order to be as hardware-agnostic as

possible, we also provide FLOPs values instead of speedup results, since these are highly dependent on the devices used.

4.1 Implementation

We test our method using the SRGAN and EDSR networks. The original SRGAN uses PReLU layers. However, due to compatibility issues, we substituted all of the PReLU layers with ReLU layers, which does not seem to have degraded its measures at all. We use the DIV2K dataset [47] for training, which consists of 800 images in the training set, 100 images in the validation set, and 100 images in the test set. All of the 1000 are 2K resolution (they have 2K pixels in at least one of the spatial dimensions) [47]. During training, we crop a 96x96 patch from the image, and we perform data augmentation by randomly rotating the image horizontally. We also report our results on other widely used datasets, such as Set5 [4], Set14 [51], Urban100 [21] and BSD100 [40]. For fair comparison with previous methods, we calculate PSNR by removing the n pixels around the final image, where n is the upscaling factor, and using the Y channel in the (Y, Cb, Cr) format instead of RGB format. We trained using the Adam optimizer [26] using a learning rate of 10^{-4} and dividing the it by 10 when the loss function plateaus. The Spatial SVD, Channel Pruning, and Quantization were performed using the AIMET toolkit [1].

		SRGAN	SRGAN-04	SRGAN-03	SRGAN-02
	GFLOPs	36.47	13.03	8.53	5.71
Set 5	**PSNR**	29.19	28.85	28.48	28.46
	SSIM	0.85	0.84	0.83	0.83
	LPIPS	0.11	0.14	0.16	0.19
Set 14	**PSNR**	26.27	25.97	25.86	25.76
	SSIM	0.73	0.72	0.72	0.72
	LPIPS	0.21	0.25	0.26	0.29
Urban100	**PSNR**	23.51	23.18	22.99	22.85
	SSIM	0.71	0.70	0.70	0.67
	LPIPS	0.21	0.26	0.27	0.31
BSDS100	**PSNR**	25.41	25.27	25.21	25.19
	SSIM	0.68	0.68	0.67	0.67
	LPIPS	0.28	0.33	0.34	0.38

Table 1: Quantitative results on channel pruning different degrees of SRGAN with 16 blocks. GFLOPs calculations under 128x128 input image with three channels. Our method achieves 6.3x less GFLOPs on SRGAN by losing less than 0.75dB PSNR accross all datasets.

4.2 Quantitative Results

Table 1 summarises the result of running our method on SRGAN. The first row shows which architecture was used: SRGAN-0X corresponds to the SRGAN architecture after performing SVD decomposition and channel pruning such that approximately 0.X of the original FLOPs are remaining (i.e. SRGAN-04 means the algorithm aimed at 40% of FLOPs remaining). The second row shows the amount of GFLOPs needed to run the network on a 128x128x3 input image. The next 4 rows shows the results for the Set5, Set14, Urban100 and BSDS100 datasets, for PSNR (in dB), SSIM, and LPIPS. As it can be seen in the table, using this method, we can reduce the total FLOPs from 36.47G, to 5.71G, a reduction of 6.3x, whilst still staying within 0.75dB in PSNR. Reducing the total FLOPs by about 3x corresponds to only 0.3dB in PSNR lost.

Also as expected, the higher is the compression, the more there is going to be a degradation in the metrics. Most notably, the LPIPS metric is the one that suffered the most, with variation of up to 0.1 in the metric. We found that qualitatively this can noticeably affect the image quality, which is discussed in Section 4.3. Therefore the trade-off between speedup and quality has to be taken into consideration depending on the application.

		EDSR	**EDSR-04**	**EDSR-03**	**EDSR-02**
Set 5	**PSNR**	30.60	29.69	29.42	29.42
	SSIM	0.89	0.88	0.88	0.87
	LPIPS	0.18	0.19	0.20	0.20
Set 14	**PSNR**	25.26	25.31	25.00	24.99
	SSIM	0.77	0.75	0.75	0.74
	LPIPS	0.29	0.31	0.31	0.32
Urban100	**PSNR**	23.21	22.08	22.00	21.72
	SSIM	0.76	0.73	0.73	0.72
	LPIPS	0.24	0.28	0.29	0.3
BSDS100	**PSNR**	25.66	25.16	25.13	24.97
	SSIM	0.73	0.72	0.72	0.72
	LPIPS	0.38	0.40	0.41	0.41

Table 2: Results on Running our method on EDSR.

In order to show that our model can be used with any superresolution architecture, we also tested it in EDSR [34]. The results are compiled in Table 2. As it can be seen, EDSR suffer considerably less degradation in LPIPS when compared to SRGAN, ranging from 0.02 to 0.06, whereas SRGAN suffered up to 0.1. This can be caused due to the difference in architecture, and the way they are trained: EDSR uses only the $L1$ norm when training. It is not surprising that EDSR then suffers considerably more degradation with respect to PSNR, which peaks at more than 1dB loss.

As it was argued in [30], PSNR does not tell the whole story, since high PSNR can be achieved with significant blur. Therefore, we have included the qualitative results in the next section.

4.3 Qualitative Results

Figure 3 shows our method on 4 images of the Set 5 dataset. From left to right, the columns correspond to the original HR image, the SRGAN image, the bicubic image, the 40% degraded, and the 20% degraded SRGANs. The compression adds little degradation in image quality when 40% is used. However, there is a noticeable blur introduced by the model when 20% degradation is used. This is not unexpected, since a compression of this magnitude would be equivalent to using only 50% of the FLOPs available for SRGAN with 1 residual block only. It is still significantly better than the simple case of bicubic interpolation.

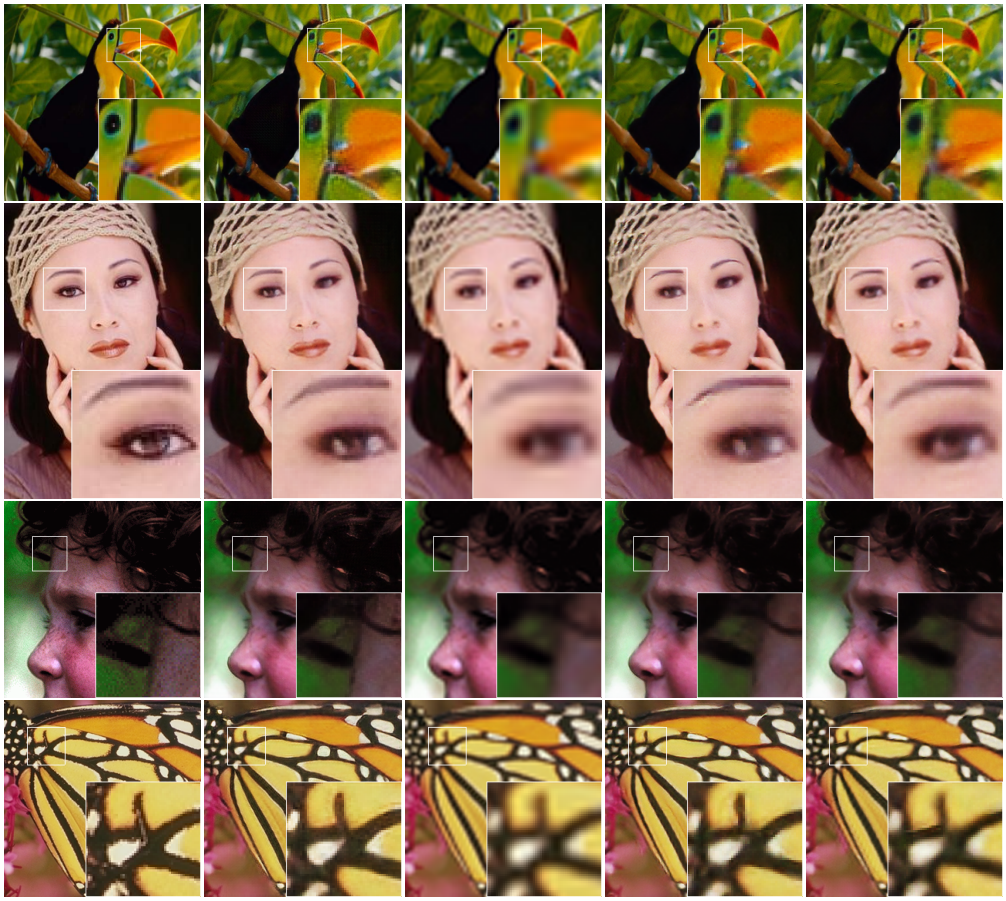


Fig. 3: From left to right column: original HR image; SRGAN; bicubic interpolation; SRGAN-04; SRGAN-02.

Figure 4 shows the same 4 images from the Set5 dataset tested using EDSR. From left column to the right are the images corresponding to the original HR image, the image produced by EDSR, the bicubic interpolation image, then EDSR-04, and EDSR-02. Differently than the results for SRGAN, little difference can be seen between EDSR-04 and EDSR-02, and both look very close to the original EDSR, with a slightly more noticeable blur. All of the neural networks generated images look considerably better than the bicubic interpolated one.

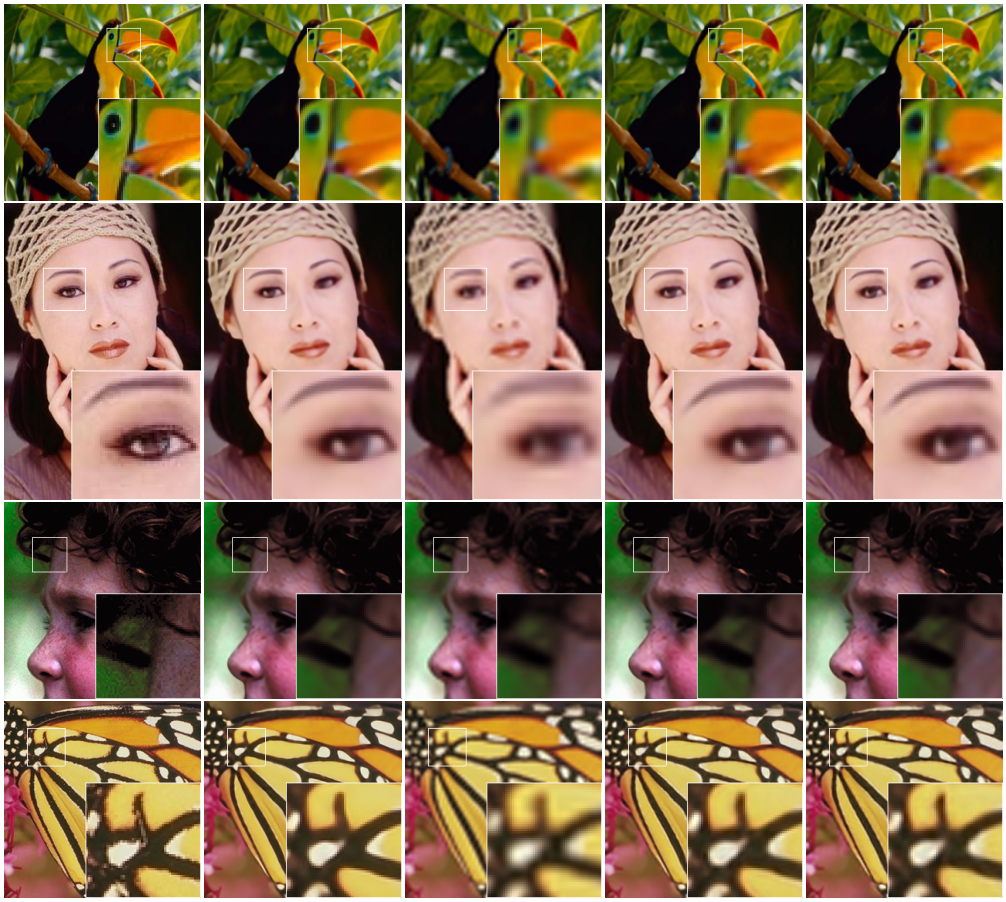


Fig. 4: From left to right column: original HR image; EDSR; bicubic interpolation; EDSR-04; EDSR-02.

4.4 Real Time reduction

We tested the networks on commodity hardware to obtain the amount of speedup in different environments. We used the SNPE toolkit [2] to convert a Tensorflow model to an intermediate representation called DLC, which is then deployed

to the Qualcomm Snapdragon processor (SM8150 and SM8350), which has an NPU optimized for running 8-bit quantized convolutional neural networks. We also tested the results on the NVidia GeForce RTX 2080 GPU. The results are summarized on Table 3.

	FLOP	**NPU1 (ms)**	**GPU (ms)**	**NPU2 (ms)**	**GPU (ms)**
	-	224x224		128x128	
SRGAN	100%	558.5	67.6	18.0	22.7
SRGAN-04	35.7%	120.7	30.9	8.9	16.2
SRGAN-03	23.4%	115.8	29.9	8.8	15.5
SRGAN-02	15.6%	99.3	25.4	7.6	15.4

Table 3: Summary of results when running a low resolution images with sizes 128x128 and 224x224 pixels, each with three colour channels, upscaling in both height and width by 4x, resulting in 512x512 and 896x896 images. The result for each hardware is the average after running 10 images with batch size equals to 1. NPU1 corresponds to SM8150 and NPU2 corresponds to SM8350.

From the table, it is interesting to note that the speedup does not necessarily correspond to FLOPs reduction. This can be because of many reasons: the time taken to run these models do not depend exclusively on number of FLOPs, but also on layers that manage memory, such as the PixelShuffler layer; also depending on how the hardware is optimized, it might work better with layers that has channels as a power of 2 due to memory access. Therefore, due to the high complexity of running these models on hardare, it is expected that the speedup does not reliably follow the reduction in number of FLOPs.

There is a natural diminishing returns when it comes to image degradation and neural network acceleration. We can achieve 5.6x speedup between the original model and SRGAN-04. However, only a marginal speedup of 20% is noticed from SRGAN-04 to SRGAN-02, even though SRGAN-02 requires 50% of the FLOPs compared to SRGAN-04. Depending on the use scenario, it might be better to sacrifice a little bit of the performance on hardware in order to have a slightly better image quality.

Also note that speedup is highly dependent on the size of the input image. For images that are 224x224 pixels, the maximum speedup in GPU is 2.6x, and the maximum speedup in the NPU is 5.6x. However, when smaller images are fed into the network, the maximum speedup in GPU is of only 1.5x, and in the NPU is of 2.4x. This is most likely due to other factors such as memory access.

Lastly, Table 4 shows the results of running the compressed models of EDSR on the Qualcomm Snapdragon processor (SM8350). From the table, it can be seen that at the maximum compression, our model has only 12% of the total number of MAC required by the original EDSR model, and it runs 7.22x faster on hardware. The slightly less compressed EDSR-04 achieves a speedup of 5.9x.

	MAC (G)	NPU (ms)
EDSR	41.7	35.4
EDSR-04	11.3	6.0
EDSR-03	8.6	5.7
EDSR-02	5.0	4.9

Table 4: Results on running an image with size 128x128 with 3 channels on the Snapdragon SM8350.

As expected, real speedup is not 100% correlated with MAC reduction due to other factors. It is interesting to note though that a speedup of 7.22x can be achieved on hardware without major differences in the quality of the superresolved images.

4.5 Time comparison with previous methods

When using channel pruning, previous methods focus solely on MAC reduction. Most of them use a Channel Selection operator, which slices the feature maps on the fly. This operator does not add MAC requirements in the hardware, giving the illusion of running quicker. However they do take a considerable amount of time to run on accelerators, because accelerators are optimized for MAC operations, whereas channel selection layers require a significant amount of memory manipulation.

To illustrate how our method is better, we ran a simple MNIST models for comparison: one of them using the Channel Selection operator, and the other one using our SVD approach. They are composed of 4 convolutional layers, each with input channels and output channels equal to 128 - apart from the first layers which has input channels equal to 1. The channel selection layers selects 64 channels before each convolution. The SVD model substitutes each layer with two consecutive convolutions with the intermediate channel size being 96. They were all run on the Qualcomm SM8350 chip. The results are shown in Table 5.

	MAC (G)	NPU (ms)	CPU (ms)
Ours	0.28	0.57	14.1
Channel Selection [17, 35]	0.28	3.82	14.6
Baseline	0.45	0.61	20.9

Table 5: Results on comparing the prevalent channel selection operator with our method on the Snapdragon SM8350.

As it can be seen, even though our method and the Channel Selection method [17, 35] have the same MAC requirements, our method is considerably quicker

using the NPU because it does not require the need to manipulate memory, which is not as optimized as convolutional operators. When only the CPU is used, the channel selection method has comparable results to ours, since the convolutional operators are less optimized in this setting.

4.6 Ablation Study

As a further step to demonstrate the effectiveness of using knowledge distillation in order to recover the accuracy degradation caused by the compression techniques, we have compiled Table 6. It can be seen that using Knowledge Distillation (KD) is better in all measures, in all networks, and in all datasets. Interestingly, the effect of Knowledge Distillation seems larger when using SRGAN-04 than when using SRGAN-02. This is expected since the architectural differences between SRGAN-04 and the original model is lower than SRGAN-02 and the original model.

		Set 5			**Set 14**		
	KD	**PSNR**	**SSIM**	**LPIPS**	**PSNR**	**SSIM**	**LPIPS**
SRGAN	N/A	29.19	0.85	0.11	26.27	0.73	0.21
SRGAN-02	Yes	28.46	0.83	0.19	25.76	0.72	0.29
SRGAN-02	No	27.93	0.82	0.19	25.39	0.71	0.29
SRGAN-03	Yes	28.48	0.83	0.16	25.86	0.72	0.25
SRGAN-03	No	27.32	0.79	0.15	25.01	0.68	0.25
SRGAN-04	Yes	28.85	0.84	0.14	25.97	0.72	0.24
SRGAN-04	No	27.86	0.81	0.15	25.16	0.69	0.24

Table 6: Ablation Study comparing the effect of using Knowledge Distillation to recover accuracy after compression. The same compressed model was retrained using KD and without KD. The results show that KD is a useful technique to recover degradation after compression.

4.7 Comparing with different Structure

Many previous papers have used KD to train a shallower variation of the base model (using ResNet50 as the teacher and ResNet34 as the student). We argue that the better solution is to use KD on a compressed version of the base model instead of a shallower model. We investigated training SRGAN with only 1 block, instead of the normal 16 blocks, and the results are in Table 7.

It can be seen that using SRGAN-02 is beneficial to using SRGAN with only one block, since they have comparable results, but SRGAN-02 requires less than a third of the FLOPs. This suggests that for superresolution tasks, it is better to compress a model using knowledge distillation as a means to recover accuracy than to train a smaller one.

	Blocks	GFLOPs	Param (10^6)	Set 5 PSNR	Set 5 SSIM	Set 5 LPIPS
SRGAN	16	111.68	1.55	29.19	0.85	0.11
SRGAN	1	55.91	0.44	28.11	0.82	0.17
SRGAN-02	16	17.47	0.13	28.46	0.83	0.19

Table 7: Comparing SRGAN with only one block with compressed version of SRGAN. As it can be seen, the compressed version of SRGAN achieves slightly better results whilst using less than a third of the FLOPs required by the SRGAN with only 1 block. FLOPs assume input image size of 224x224

5 Conclusion

We introduced in this paper a method to compress superresolution neural networks to accelerate inference in commodity hardware. We showed that significant compression is possible without catastrophic image degradation. We also showed that it is beneficial to perform a series of model compression with knowledge distillation as a way of recovering image degradation. We tested our model on commodity hardware and achieved 5.6x speed on SRGAN and 7.2x speedup on EDSR.

References

1. Ai model efficiency toolkit. `https://quic.github.io/aimet-pages/index.html`, accessed: 2021-06-25
2. Snapdragon neural processing engine sdk. `https://developer.qualcomm.com/docs/snpe/overview.html`, accessed: 2021-06-25
3. Anwar, S., Sung, W.: Coarse pruning of convolutional neural networks with random masks (2016)
4. Bevilacqua, M., Roumy, A., Guillemot, C., Alberi-Morel, M.L.: Low-complexity single-image super-resolution based on nonnegative neighbor embedding (2012)
5. Bianco, S., Cadene, R., Celona, L., Napoletano, P.: Benchmark analysis of representative deep neural network architectures. IEEE Access **6**, 64270–64277 (2018)
6. Buciluǎ, C., Caruana, R., Niculescu-Mizil, A.: Model compression. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 535–541 (2006)
7. Denton, E., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R.: Exploiting linear structure within convolutional networks for efficient evaluation. arXiv preprint arXiv:1404.0736 (2014)
8. Dong, C., Loy, C.C., He, K., Tang, X.: Image super-resolution using deep convolutional networks. IEEE transactions on pattern analysis and machine intelligence **38**(2), 295–307 (2015)
9. Dong, C., Loy, C.C., Tang, X.: Accelerating the super-resolution convolutional neural network. In: European conference on computer vision. pp. 391–407. Springer (2016)
10. Gao, Q., Zhao, Y., Li, G., Tong, T.: Image super-resolution using knowledge distillation. In: Asian Conference on Computer Vision. pp. 527–541. Springer (2018)
11. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 580–587 (2014)
12. Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks. arXiv preprint arXiv:1406.2661 (2014)
13. Gou, J., Yu, B., Maybank, S.J., Tao, D.: Knowledge distillation: A survey. International Journal of Computer Vision **129**(6), 1789–1819 (2021)
14. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
15. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural networks. arXiv preprint arXiv:1506.02626 (2015)
16. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
17. He, Y., Zhang, X., Sun, J.: Channel pruning for accelerating very deep neural networks. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 1389–1397 (2017)
18. He, Z., Dai, T., Lu, J., Jiang, Y., Xia, S.T.: Fakd: Feature-affinity based knowledge distillation for efficient image super-resolution. In: 2020 IEEE International Conference on Image Processing (ICIP). pp. 518–522 (2020). https://doi.org/10.1109/ICIP40778.2020.9190917

19. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)
20. Hou, Z., Kung, S.Y.: Efficient image super resolution via channel discriminative deep neural network pruning (2020)
21. Huang, J.B., Singh, A., Ahuja, N.: Single image super-resolution from transformed self-exemplars. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 5197–5206 (2015)
22. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research **18**(1), 6869–6898 (2017)
23. Hui, Z., Wang, X., Gao, X.: Fast and accurate single image super-resolution via information distillation network. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 723–731 (2018)
24. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. arXiv preprint arXiv:1405.3866 (2014)
25. Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D.T., Jammalamadaka, N., Huang, J., Yuen, H., et al.: A study of bfloat16 for deep learning training. arXiv preprint arXiv:1905.12322 (2019)
26. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
27. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems **25**, 1097–1105 (2012)
28. Lebedev, V., Lempitsky, V.: Fast convnets using group-wise brain damage. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 2554–2564 (2016)
29. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Advances in neural information processing systems. pp. 598–605 (1990)
30. Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., et al.: Photo-realistic single image super-resolution using a generative adversarial network. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 4681–4690 (2017)
31. Lee, R., Venieris, S.I., Dudziak, L., Bhattacharya, S., Lane, N.D.: Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In: The 25th Annual International Conference on Mobile Computing and Networking. pp. 1–16 (2019)
32. Lei, S., Shi, Z., Zou, Z.: Coupled adversarial training for remote sensing image super-resolution. IEEE Transactions on Geoscience and Remote Sensing **58**(5), 3633–3643 (2019)
33. Li, F., Zhang, B., Liu, B.: Ternary weight networks. arXiv preprint arXiv:1605.04711 (2016)
34. Lim, B., Son, S., Kim, H., Nah, S., Mu Lee, K.: Enhanced deep residual networks for single image super-resolution. In: Proceedings of the IEEE conference on computer vision and pattern recognition workshops. pp. 136–144 (2017)
35. Liu, Z., Mu, H., Zhang, X., Guo, Z., Yang, X., Cheng, K.T., Sun, J.: Metapruning: Meta learning for automatic neural network channel pruning. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 3296–3305 (2019)
36. Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., Zhang, C.: Learning efficient convolutional networks through network slimming. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 2736–2744 (2017)

37. Lugmayr, A., Danelljan, M., Timofte, R.: Ntire 2020 challenge on real-world image super-resolution: Methods and results. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. pp. 494–495 (2020)
38. Luo, J.H., Zhang, H., Zhou, H.Y., Xie, C.W., Wu, J., Lin, W.: Thinet: pruning cnn filters for a thinner net. IEEE transactions on pattern analysis and machine intelligence **41**(10), 2525–2538 (2018)
39. Ma, Y., Xiong, H., Hu, Z., Ma, L.: Efficient super resolution using binarized neural network. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. pp. 0–0 (2019)
40. Martin, D., Fowlkes, C., Tal, D., Malik, J.: A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001. vol. 2, pp. 416–423. IEEE (2001)
41. Mirzadeh, S.I., Farajtabar, M., Li, A., Levine, N., Matsukawa, A., Ghasemzadeh, H.: Improved knowledge distillation via teacher assistant. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 5191–5198 (2020)
42. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: Imagenet classification using binary convolutional neural networks. In: European conference on computer vision. pp. 525–542. Springer (2016)
43. Ren, H., El-Khamy, M., Lee, J.: Ct-srcnn: Cascade trained and trimmed deep convolutional neural networks for image super resolution. In: 2018 IEEE Winter Conference on Applications of Computer Vision (WACV). pp. 1423–1431. IEEE (2018)
44. Rigamonti, R., Sironi, A., Lepetit, V., Fua, P.: Learning separable filters. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 2754–2761 (2013)
45. Romero, A., Ballas, N., Kahou, S.E., Chassang, A., Gatta, C., Bengio, Y.: Fitnets: Hints for thin deep nets. arXiv preprint arXiv:1412.6550 (2014)
46. Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A.P., Bishop, R., Rueckert, D., Wang, Z.: Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1874–1883 (2016)
47. Timofte, R., Agustsson, E., Van Gool, L., Yang, M.H., Zhang, L.: Ntire 2017 challenge on single image super-resolution: Methods and results. In: Proceedings of the IEEE conference on computer vision and pattern recognition workshops. pp. 114–125 (2017)
48. Wang, N., Choi, J., Brand, D., Chen, C.Y., Gopalakrishnan, K.: Training deep neural networks with 8-bit floating point numbers. arXiv preprint arXiv:1812.08011 (2018)
49. Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y., Dong, C., Qiao, Y., Change Loy, C.: Esrgan: Enhanced super-resolution generative adversarial networks. In: Proceedings of the European Conference on Computer Vision (ECCV) Workshops. pp. 0–0 (2018)
50. Xin, J., Wang, N., Jiang, X., Li, J., Huang, H., Gao, X.: Binarized neural network for single image super resolution. In: European Conference on Computer Vision. pp. 91–107. Springer (2020)
51. Zeyde, R., Elad, M., Protter, M.: On single image scale-up using sparse-representations. In: International conference on curves and surfaces. pp. 711–730. Springer (2010)

52. Zhang, X., Zhou, X., Lin, M., Sun, J.: Shufflenet: An extremely efficient convolutional neural network for mobile devices. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 6848–6856 (2018)
53. Zhang, Y., Li, K., Li, K., Wang, L., Zhong, B., Fu, Y.: Image super-resolution using very deep residual channel attention networks. In: Proceedings of the European conference on computer vision (ECCV). pp. 286–301 (2018)
54. Zhang, Y., Tian, Y., Kong, Y., Zhong, B., Fu, Y.: Residual dense network for image super-resolution. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 2472–2481 (2018)

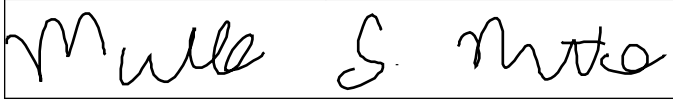
Statement of Authorship for joint/multi-authored papers for PGR thesis

To appear at the end of each thesis chapter submitted as an article/paper

The statement shall describe the candidate's and co-authors' independent research contributions in the thesis publications. For each publication there should exist a complete statement that is to be filled out and signed by the candidate and supervisor **(only required where there isn't already a statement of contribution within the paper itself).**

Title of Paper	A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks
Publication Status	□ Published □ Accepted for Publication □ Submitted for Publication X Unpublished and unsubmitted work written in a manuscript style
Publication Details	*Gennari do Nascimento, Marcelo, Karthik Vijayan, Sunando Sengupta, Alexandros Neopythou, and Eric Sommerlade. "A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks." In manuscript format.*

Student Confirmation

Student Name:	Marcelo Gennari do Nascimento		
Contribution to the Paper	• Conception of most of the ideas, with collaboration from Alexandros, Sunando, Karthik and Eric. • Designed, and wrote all of the code • Performed all the experiments, apart from acceleration experiments which were performed by Karthik • Created all of the tables and diagrams • Wrote the entire manuscript		
		Date	12/09/2022

Supervisor Confirmation

By signing the Statement of Authorship, you are certifying that the candidate made a substantial contribution to the publication, and that the description described above is accurate.

Supervisor name and title: Professor Victor Adrian Prisacariu			
Supervisor comments The description on contributions is accurate			
Signature		Date	12/09/2022

This completed form should be included in the thesis, at the end of the relevant chapter.

6.5 Conclusion

In this chapter we introduced *A Knowledge Distillation Approach to Pruning and Accelerating Superresolution Neural Networks*. Our previous papers have focused on using quantisation for compression, and relying on the design of specialized hardware to run the methods using low power. This chapter provided a method to compress a network in order to run on hardware that already exists.

We introduced a pipeline consisting of three compression methods: SVD decomposition, which breaks a convolutional layer into two, and allows for each convolution to have an "inside" channel value that can be changed without affecting the rest of the network, thus solving the "branching" problem; Channel Pruning, which is a dense-preserving method to decrease the number of FLOPs required in the CNN; 8-bit quantization, which is the most ubiquitously supported low precision methods in current hardware. We proposed to use knwledge distillation (KD) to bridge the accuracy gap between each of these compression steps.

We show that by using KD after each step we are able to recover more accuracy than using KD after the three steps are done at once. By using this pipeline, we were able to achieve a 5× speedup on SRGAN and a 7.2× speedup on EDSR.

7 Discussion and Conclusion

Contents

7.1 Discussion 112
7.2 Achievements 112
7.3 Future Work 114

This thesis has presented a perspective on accelerating CNNs for a variety of different applications. In Chapters 3 to 5 we presented cases for quantization: we showed techniques that can be used for inference, storage, using unlabelled data, and training. We also extended this work to the case in which not all layers have to be quantized to the same bitwidth and presented a way of finding optimal distributions. In Chapter 6 we delved into approaches that work on commodity hardware, such as knowledge distillation combined with tensor decomposition, pruning, and 8-bit quantization.

In this chapter, we will present a discussion and general overview of what has been introduced. We will also delve into their limitations, an overview of the achievements and contributions of this thesis, and scope for potential future work.

7.1 Discussion

It is estimated that the human brain requires around 20W to function [146], which includes not only its perception and proprioception systems, but all of the other functions of the body. A ResNet50, used for classification of a 224x224 image only was estimated to require 13W to run [28]. Although it is obvious that a lot of the difference is due to the way each method is processed (brains are organic based, with neurons working in a model much closer to Spiking Neural Networks [147], which indeed can be less power hungry when using neuromorphic hardware, rather than the Convolutional Neural Networks used in today's computer vision research), at least this provides evidence that much more efficient ways of performing this task exist and can be found.

This thesis has delved into the many methods available to compress neural networks with the final goal of solving computer vision tasks using minimal amount of compute. In the next sections, we will elaborate on the achievements and contributions presented so far, and possible future research lines to achieve this goal.

7.2 Achievements

We have presented four papers that expand the literature on accelerating convolutional neural networks. In this way, we have extensively explored and contributed to the community in four closely related areas:

Quantization for Quick Inference Our first paper extended the literature by providing a method that leverages the block floating point format for quick inference. By sharing exponents across blocks channel-wise, we provide a simple and effective way of balancing the trade-off between accuracy degradation and network compression. We have shown results for a variety of different compression tasks, such as compressing for storage, compressing using unlabelled data, and retraining for compression. We extensively tested our model in large datasets such as ImageNet and large models such as ResNet-50.

Non-Uniform Quantization Our second paper has introduced the idea of using Bayesian Optimization (BO) with Gaussian Process priors to find optimal distribution of bitwidths through the layers of a CNN. Since each layer can be compressed to different extents, our algorithm searches for the configuration that achieves the highest accuracy whilst limiting the amount of memory utilization. This is in contrast to previous papers that use approaches that are not as sample-efficient as BO, such as Reinforcement Learning (RL).

Quantization for Quick Training Our third paper provided a framework to perform neural network training and backpropagation by quantizing not only the weights and feature maps, but also the gradients. This is ultimately the goal of quantization, but it is extremely difficult due to the fact that gradients generally need a higher degree of granularity and utilize a large scale of values. We extended the idea of channel-wise block floating point to 2 dimensions, which transforms the quantization procedure transpose invariant. This allows for backpropagation to be carried without multiple quantization calls. We followed this with extensive tests on ImageNet, where we could train AlexNet using only 4 bits, and ResNet18 using only 5 bits with virtually no accuracy degradation.

Accelerating CNN for Commodity Hardware Our fourth paper focused on approaches to run quick inference on current commodity hardware, which do not support the latest advances in low-bit quantization. We therefore took advantage of other methods, such as SVD decomposition, Channel Pruning and 8-bit quantization. In order to apply all of these compression methods, we bridge each one of them by using Knowledge Distillation (KD), which helps the network to keep its accuracy throughout the pipeline. We show through extensive experimentation that bridging the compression with KD is much better than first performing the compression and then distilling.

7.3 Future Work

The field of finding effective compressed networks is vast, and there are still many methods and research lines that this thesis could not cover. Next, we list a couple of ideas that are worth investigating in subsequent work.

Adaptation for Transformer Networks

A considerable amount of work has been done on accelerating Convolutional Neural Networks. Parallel to the advances in Computer Vision, the conception of the Transformer [148] has achieved impressive results in Natural Language Processing. Naturally, researchers started to adapt these models to work on images as well. Since then it has also achieved impressive results on computer vision tasks, such as the Vision Transformer [149]. The most notable disadvantage of these models is that they win on scale, which means that they need to train on a bigger dataset to achieve results comparable to CNNs. More modern ViTs have achieved this goal and are comparable in accuracy and FLOPs to ResNet models [150]. Since then, there has been a rise in the idea that Transformers may surpass convolutional neural networks for computer vision tasks.

The debate on which model to use is still ongoing and there are many arguments for both sides [151]. However there has been only limited research on how to make these models more compact.

Hardware-Aware Search

As it was argued in Chapter 6, using FLOPs as a proxy for computational load is not the most accurate measure. Very often, models with higher FLOPs perform better in certain hardware than models with smaller FLOPs. This has also been demonstrated in literature, for example with the work of [45].

It is also important to note that different hardware chip is optimized for different operations. With the rise of specially designed "AI" chips (NPUs) from big manufacture vendors, there is a discrepancy on support for each of them. For regular CPUs, they thrive in matrix multiplications specially when aligned with

SIMD instructions such as AVX-512, which is why it is often implemented by first flattening the input by using `im2col` operations and then using the optimized matrix multiplication routine [152], which means that certain sparse algorithms might make it run quicker. Often NPUs are already optimized for 8-bit convolutions, and use a direct convolution approach which avoid costly memory operations, which also means that they might perform worse on operations such as `gather` and vector shuffling [152].

There is a need to automatically design neural networks to solve computer vision tasks and are optimized specifically for a target hardware. The way this is done today is by designing a single network with low FLOPs and using it in a multitude of hardware architectures. The ideal scenario would be to have multiple CNNs, each with similar accuracy but optimized to run quicker on each target hardware, and more research needs to be done to achieve this goal efficiently.

The Impact of Rounding Policies

Whenever a network is quantized, there needs to be a rounding procedure. As we explained in Chapter 5, stochastic rounding has proved to be an effective way of avoiding stagnation of neural networks. However, since it is a stochastic method, it is prone to instability. We have shown in Chapter 2 that smart rounding can achieve good results on post training quantization without the need to retrain at all [63], however the avenue of smart rounding for training neural networks has not been explored yet.

This goes hand in hand with the idea of *dithering*. Stochastic Rounding equates to adding an unconditional uniformly sampled value and flooring the result. There is an argument that the uniform distribution might not be the optimal distribution for rounding: it does place a higher chance of rounding to the nearest value, which equates to lowest distance to original network in weight-space, but it also does not take into account neighbouring values, which is essential for effective quantization as shown in [63]. Also, previous studies has shown the effectiveness of other distributions [153]. There needs to be more research on finding deterministic ways

of rounding gradients, how to effective round them for optimal training, and on conditional distributions for stohcastic rounding.

Joint Speed-Accuracy Optimization

Very related to the idea of Hardware-Aware Search is the multi-objective task of optimizing jointly speed and accuracy. Most previous papers that take into account more than one parameter simplify the search by using a hand-designed score function which takes into account all of the parameter and returns a single score value. This approach seems to work, but it requires the user to hand-pick hyperparameters and decide on trade-offs before the search has even started - for example, decide that sacrificing 1% of accuracy is worth it if the speedup is 1.5x or higher.

Another way of potentially solving this problem is by a joint optimization problem. For example, Bayesian Optimization using Gaussian Processes provides a useful framework for Multi-Objective search [154–156]. These tasks though assume that all of the objectives are being evaluated simultaneously. However, computing speed is way more cost effective than computing the accuracy of a neural network, since retraining is not necessary. To the best of our knowledge, there is no method that takes advantage of multi-objective with different evaluation costs applied to neural architecture search. This is a promising frontier to explore, since it aligns the objectives of speed and accuracy by using a cheap proxy for exploration.

References

[1] Irwin Sobel. "An Isotropic 3x3 Image Gradient Operator". In: *Presentation at Stanford A.I. Project 1968* (Feb. 2014).

[2] T. Lindeberg. "Scale-space for discrete signals". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12.3 (1990), pp. 234–254.

[3] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698.

[4] D.G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2.

[5] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[6] Haohan Wang and Bhiksha Raj. "On the origin of deep learning". In: *arXiv preprint arXiv:1702.07800* (2017).

[7] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[9] Olga Russakovsky et al. "Detecting avocados to zucchinis: what have we done, and where are we going?" In: *International Conference on Computer Vision (ICCV)*. 2013.

[10] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252.

[11] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[12] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1409.1556.

[13] Christian Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9.

[14] Ross Girshick. "Fast R-CNN". In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1440–1448.

[15] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

[16] Kaiming He et al. "Mask R-CNN". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988.

[17] Alexander Toshev and Christian Szegedy. "DeepPose: Human Pose Estimation via Deep Neural Networks". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 1653–1660.

[18] Christian Ledig et al. "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 105–114.

[19] Fuzhen Zhuang et al. "A comprehensive survey on transfer learning". In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76.

[20] *Snapdragon Neural Processing Engine SDK*. https://developer.qualcomm.com/docs/snpe/overview.html. Accessed: 2021-06-25.

[21] *Arm Ethos-U65 microNPU*. https://armkeil.blob.core.windows.net/developer/Files/pdf/arm-ethos-u65-product-brief-v2.pdf. Accessed: 2022-04-25.

[22] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[23] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Apr. 2018.

[24] Zheng Qin et al. "Fd-Mobilenet: Improved Mobilenet with a Fast Downsampling Strategy". In: *2018 25th IEEE International Conference on Image Processing (ICIP)*. 2018, pp. 1363–1367.

[25] Xiangyu Zhang et al. "Shufflenet: An extremely efficient convolutional neural network for mobile devices". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 6848–6856.

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

[27] Eugenio Culurciello. *Neural Network Architectures*. Mar. 2017. URL: https://towardsdatascience.com/neural-network-architectures-156e5bad51ba.

[28] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. "An Analysis of Deep Neural Network Models for Practical Applications". In: *CoRR* abs/1605.07678 (2016). arXiv: 1605.07678. URL: http://arxiv.org/abs/1605.07678.

[29] Jimmy Ba and Rich Caruana. "Do Deep Nets Really Need to be Deep?" In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/ea8fcd92d59581717e06eb187f10666d-Paper.pdf.

[30] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[31] Yann Lecun, John Denker, and Sara Solla. "Optimal Brain Damage". In: vol. 2. Jan. 1989, pp. 598–605.

[32] Qualcomm. URL: https://www.qualcomm.com/system/files/document/files/prod_brief_qcom_sd888_5g_1_0.pdf.

[33] Shibo Wang and Pankaj Kanware. *BFloat16: The secret to high performance on Cloud TPUs*. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. Accessed: 2020-03-08.

[34] Jonghoon Jin, Aysegul Dundar, and Eugenio Culurciello. "Flattened Convolutional Neural Networks for Feedforward Acceleration". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: https://arxiv.org/abs/1412.5474.

[35] Min Wang, Baoyuan Liu, and Hassan Foroosh. "Factorized Convolutional Neural Networks". In: *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*. 2017, pp. 545–553.

[36] Gao Huang et al. "Densely Connected Convolutional Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[37] Barret Zoph and Quoc V. Le. *Neural Architecture Search with Reinforcement Learning*. 2016. arXiv: 1611.01578 [cs.LG].

[38] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[39] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018).

[40] Hieu Pham et al. "Efficient neural architecture search via parameter sharing". In: *arXiv preprint arXiv:1802.03268* (2018).

[41] Dilin Wang et al. "Attentivenas: Improving neural architecture search via attentive sampling". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 6418–6427.

[42] Jiahui Yu et al. "Bignas: Scaling up neural architecture search with big single-stage models". In: *European Conference on Computer Vision*. Springer. 2020, pp. 702–717.

[43] Xiaoliang Dai et al. "FBNetV3: Joint architecture-recipe search using predictor pretraining". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 16276–16285.

[44] Han Cai et al. "Once-for-all: Train one network and specialize it for efficient deployment". In: *arXiv preprint arXiv:1908.09791* (2019).

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

[46] Bichen Wu et al. "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[47] IEEE Standards Comittee. *IEEE standard for binary floating-point arithmetic*. Note: Standard 754–1985. New York: Institute of Electrical and Electronics Engineers, 1985.

[48] William Dally. "High-Performance Hardware for Machine Learning". In: 2015.

[49] Mark Horowitz. "Computing's Energy Problem (and what we can do about it)". In: 2014. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6757323`.

[50] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[51] Penghang Yin et al. "Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets". In: *International Conference on Learning Representations*. 2019. URL: `https://openreview.net/forum?id=Skh4jRcKQ`.

[52] Amir Gholami et al. "A survey of quantization methods for efficient neural network inference". In: *arXiv preprint arXiv:2103.13630* (2021).

[53] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1510.00149.

[54] Wenlin Chen et al. "Compressing Neural Networks with the Hashing Trick". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 2285–2294. URL: https://proceedings.mlr.press/v37/chenc15.html.

[55] Matthieu Courbariaux and Yoshua Bengio. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *ArXiv* abs/1602.02830 (2016).

[56] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *ECCV*. 2016.

[57] Fengfu Li, Bo Zhang, and Bin Liu. "Ternary weight networks". In: *arXiv preprint arXiv:1605.04711* (2016).

[58] Chenzhuo Zhu et al. "Trained ternary quantization". In: *arXiv preprint arXiv:1612.01064* (2016).

[59] Dongqing Zhang et al. "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks". In: *European Conference on Computer Vision (ECCV)*. 2018.

[60] Andrew Howard et al. "Searching for MobileNetV3". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019.

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

[62] Alexander Finkelstein, Uri Almog, and Mark Grobman. "Fighting quantization bias with bias". In: *arXiv preprint arXiv:1906.03193* (2019).

[63] Markus Nagel et al. "Up or down? adaptive rounding for post-training quantization". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 7197–7206.

[64] Markus Nagel et al. "Data-free quantization through weight equalization and bias correction". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1325–1334.

[65] Yaohui Cai et al. "Zeroq: A novel zero shot quantization framework". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 13169–13178.

[66] Matan Haroush et al. "The knowledge within: Methods for data-free model compression". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 8494–8502.

[67] Yoojin Choi et al. "Data-free network quantization with adversarial knowledge distillation". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2020, pp. 710–711.

[68] Xiangyu He et al. "Generative zero-shot network quantization". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 3000–3011.

[69] Shoukai Xu et al. "Generative low-bitwidth data free quantization". In: *European Conference on Computer Vision*. Springer. 2020, pp. 1–17.

[70] Xiaofan Lin, Cong Zhao, and Wei Pan. "Towards accurate binary convolutional neural network". In: *Advances in neural information processing systems* 30 (2017).

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

[72] Zhaowei Cai et al. "Deep learning with low precision by half-wave gaussian quantization". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5918–5926.

[73] Markus Höhfeld and Scott E. Fahlman. "Probabilistic rounding in neural network learning with limited precision". In: *Neurocomputing* 4.6 (1992), pp. 291–299. URL: https://www.sciencedirect.com/science/article/pii/092523129290014G.

[74] Suyog Gupta et al. "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 1737–1746.

[75] Michael Hopkins et al. "Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190052.

[76] Guandao Yang et al. "SWALP: Stochastic weight averaging in low precision training". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 7015–7024.

[77] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* 2.7 (2015).

[78] Seyed Iman Mirzadeh et al. "Improved knowledge distillation via teacher assistant". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 5191–5198.

[79] Linfeng Zhang et al. "Be your own teacher: Improve the performance of convolutional neural networks via self distillation". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 3713–3722.

[80] Jangho Kim et al. "Qkd: Quantization-aware knowledge distillation". In: *arXiv preprint arXiv:1911.12491* (2019).

[81] Steven W Smith et al. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.

[82] Roberto Rigamonti et al. "Learning separable filters". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2013, pp. 2754–2761.

[83] Gilbert W Stewart. "On the early history of the singular value decomposition". In: *SIAM review* 35.4 (1993), pp. 551–566.

[84] Francois Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (July 2017). URL: http://dx.doi.org/10.1109/CVPR.2017.195.

[85] Jianbo Guo et al. "Network decoupling: From regular to depthwise separable convolutions". In: *arXiv preprint arXiv:1808.05517* (2018).

[86] Yunpeng Chen et al. "Sharing Residual Units Through Collective Tensor Factorization To Improve Deep Neural Networks." In: *IJCAI*. 2018, pp. 635–641.

[87] Harold Hotelling. "Analysis of a complex of statistical variables into principal components." In: *Journal of educational psychology* 24.6 (1933), p. 417.

[88] Franck Mamalet and Christophe Garcia. "Simplifying convnets for fast learning". In: *International Conference on Artificial Neural Networks*. Springer. 2012, pp. 58–65.

[89] Emily L Denton et al. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in neural information processing systems*. 2014, pp. 1269–1277.

[90] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. "Speeding up convolutional neural networks with low rank expansions". In: *arXiv preprint arXiv:1405.3866* (2014).

[91] Xiangyu Zhang et al. "Accelerating very deep convolutional networks for classification and detection". In: *IEEE transactions on pattern analysis and machine intelligence* 38.10 (2015), pp. 1943–1955.

[92] Yuhui Xu et al. "Trained Rank Pruning for Efficient Deep Neural Networks". In: *arXiv preprint arXiv:1812.02402* (2018).

[93] Sajid Anwar and Wonyong Sung. "Coarse pruning of convolutional neural networks with random masks". In: (2016).

[94] Volker Strassen et al. "Gaussian elimination is not optimal". In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[95] Jianyu Huang et al. "Generating families of practical fast matrix multiplication algorithms". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 656–667.

[96] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 63–74. URL: https://doi.org/10.1145/106972.106981.

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

[98] G. Castellano, A.M. Fanelli, and M. Pelillo. "An iterative pruning algorithm for feedforward neural networks". In: *IEEE Transactions on Neural Networks* 8.3 (1997), pp. 519–531.

[99] Song Han et al. "Learning both weights and connections for efficient neural networks". In: *arXiv preprint arXiv:1506.02626* (2015).

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

[101] Hattie Zhou et al. "Deconstructing lottery tickets: Zeros, signs, and the supermask". In: *Advances in neural information processing systems* 32 (2019).

[102] Vivek Ramanujan et al. "What's hidden in a randomly weighted neural network?" In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 11893–11902.

[103] Vadim Lebedev and Victor Lempitsky. "Fast convnets using group-wise brain damage". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2554–2564.

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

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

[106] Jian-Hao Luo et al. "Thinet: pruning cnn filters for a thinner net". In: *IEEE transactions on pattern analysis and machine intelligence* 41.10 (2018), pp. 2525–2538.

[107] Martın Abadi et al. "{TensorFlow}: a system for {Large-Scale} machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[108] Paresh Kharya. *Nvidia blogs: TENSORFLOAT-32 accelerates AI training HPC UPTO 20X*. May 2020. URL: https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

[109] *Nvidia Hopper architecture in-depth*. May 2022. URL: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[110] Roman Ilin, Thomas Watson, and Robert Kozma. "Abstraction hierarchy in deep learning neural networks". In: May 2017, pp. 768–774.

[111] Chen Lei. "Deep reinforcement learning". In: *Deep Learning and Practice with MindSpore*. Springer, 2021, pp. 217–243.

[112] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: (1995).

[113] David Silver et al. "Mastering the game of go without human knowledge". In: *nature* 550.7676 (2017), pp. 354–359.

[114] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[115] Oriol Vinyals et al. "Starcraft ii: A new challenge for reinforcement learning". In: *arXiv preprint arXiv:1708.04782* (2017).

[116] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

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

[118] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[119] Hanxiao Liu et al. "Hierarchical representations for efficient architecture search". In: *arXiv preprint arXiv:1711.00436* (2017).

[120] Ahmed T Elthakeb et al. "Releq: A reinforcement learning approach for deep quantization of neural networks". In: *arXiv preprint arXiv:1811.01704* (2018).

[121] Pradnya A. Vikhar. "Evolutionary algorithms: A critical review and its future prospects". In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. 2016, pp. 261–265.

[122] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. "Designing Neural Networks Using Genetic Algorithms." In: *ICGA*. Vol. 89. 1989, pp. 379–384.

[123] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[124] Dario Floreano, Peter Dürr, and Claudio Mattiussi. "Neuroevolution: from architectures to learning". In: *Evolutionary intelligence* 1.1 (2008), pp. 47–62.

[125] Yuqiao Liu et al. "A survey on evolutionary neural architecture search". In: *IEEE transactions on neural networks and learning systems* (2021).

[126] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4780–4789.

[127] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=ByME42AqK7.

[128] Esteban Real et al. "Large-scale evolution of image classifiers". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2902–2911.

[129] Aditya Rawal and Risto Miikkulainen. "From nodes to networks: Evolving recurrent neural networks". In: *arXiv preprint arXiv:1803.04439* (2018).

[130] Peter I Frazier. "A tutorial on Bayesian optimization". In: *arXiv preprint arXiv:1807.02811* (2018).

[131] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems*. 2012, pp. 2951–2959.

[132] James Bergstra, Daniel Yamins, and David Daniel Cox. "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures". In: (2013).

[133] Rodolphe Jenatton et al. "Bayesian optimization with tree-structured dependencies". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1655–1664.

[134] Hector Mendoza et al. "Towards automatically-tuned neural networks". In: *Workshop on Automatic Machine Learning*. 2016, pp. 58–65.

[135] Andrew Wilson and Hannes Nickisch. "Kernel interpolation for scalable structured Gaussian processes (KISS-GP)". In: *International Conference on Machine Learning*. 2015, pp. 1775–1784.

[136] Jacob R Gardner et al. "Product kernel interpolation for scalable Gaussian processes". In: *arXiv preprint arXiv:1802.08903* (2018).

[137] Edward Snelson and Zoubin Ghahramani. "Sparse Gaussian processes using pseudo-inputs". In: *Advances in neural information processing systems*. 2006, pp. 1257–1264.

[138] Kirthevasan Kandasamy et al. "Neural Architecture Search with Bayesian Optimisation and Optimal Transport". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 2016–2025. URL: http://papers.nips.cc/paper/7472-neural-architecture-search-with-bayesian-optimisation-and-optimal-transport.pdf.

[139] Yang Yu. "Towards Sample Efficient Reinforcement Learning." In: *IJCAI*. 2018, pp. 5739–5743.

[140] Chao Qian et al. "On the effectiveness of sampling for evolutionary optimization in noisy environments". In: *Evolutionary computation* 26.2 (2018), pp. 237–267.

[141] Wenzhe Shi et al. "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1874–1883.

[142] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[143] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. 2019.

[144] Song Han et al. "EIE: Efficient inference engine on compressed deep neural network". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 243–254.

[145] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 740–753.

[146] Vijay Balasubramanian. "Brain power". In: *Proceedings of the National Academy of Sciences* 118.32 (2021), e2107022118. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2107022118. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2107022118.

[147] Michael Pfeiffer and Thomas Pfeil. "Deep Learning With Spiking Neurons: Opportunities and Challenges". In: *Frontiers in Neuroscience* 12 (2018). URL: https://www.frontiersin.org/article/10.3389/fnins.2018.00774.

[148] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[149] Alexey Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[150] Ze Liu et al. "Swin transformer: Hierarchical vision transformer using shifted windows". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 10012–10022.

[151] Zhuang Liu et al. "A ConvNet for the 2020s". In: *arXiv preprint arXiv:2201.03545* (2022).

[152] Evangelos Georganas et al. "Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018.

[153] Lu Xia et al. “A Simple and Efficient Stochastic Rounding Method for Training Neural Networks in Low Precision”. In: *arXiv preprint arXiv:2103.13445* (2021).

[154] Takashi Wada and Hideitsu Hino. “Bayesian optimization for multi-objective optimization and multi-point search”. In: *arXiv preprint arXiv:1905.02370* (2019).

[155] Samuel Daulton et al. “Multi-objective bayesian optimization over high-dimensional search spaces”. In: *arXiv preprint arXiv:2109.10964* (2021).

[156] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. “Parallel Bayesian Optimization of Multiple Noisy Objectives with Expected Hypervolume Improvement”. In: *Advances in Neural Information Processing Systems* 34 (2021).