DISSERTATION

submitted to the

Combined Faculty of Natural Sciences and Mathematics

of the

Ruprecht–Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Günther Schindler

born in
Eggenfelden, Bayern

Heidelberg, 2021

Compressing and Mapping Deep Neural Networks on Edge Computing Systems

Advisor: Professor Dr. Holger Fröning

Date of oral exam: ....

Abstract

Deep neural networks (DNNs) are a key technology nowadays and the main driving factor for many recent advances in Artificial Intelligence (AI) applications, including computer vision, natural language processing and speech recognition. DNNs exhibit the ability to excellently fit training data while also generalizing well to unseen data, which is especially effective when big amounts of data and ample hardware resource are available. These hardware requirements in terms of computations and memory are the limiting factor for their deployment in edge computing systems, such as handheld or head-word devices.

Enabling DNNs to be deployed on edge devices is one of the key challenges towards the next generation of AI applications, including augmented reality or enhanced interactions between humans and computers. Three major research directions have to be jointly considered for effective deployment: efficient model design, high-performance hardware as well as cooperating software frameworks. This work studies these research directions from an holistic point of view and carefully considers the impact of one directions to the others, in order to develop techniques that improve the overall deployment.

First, efficient model design through compression in form of quantization is studied, to reduce the required data representation from single-precision floating point to low-bit formats. Several quantization techniques are evaluated and a library is introduced that enables arbitrary bit combination on Central Processing Units (CPUs). The potential and implications of mapping quantized DNNs is extensively studied on mobile CPUs as well as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs).

The next part considers the limitations of quantized DNNs and proposes a compression/algorithmic co-design, targeting fast deployment on mobile CPUs while achieving high prediction quality. The proposed compression algorithm is based on an adaptive quantization function that additionally induces sparsity into the DNN. A deployment algorithm is introduced for accelerating computations by exploiting the aggressively-low and sparse data formats, created by the compression technique.

The final parts address the disadvantages of extreme forms of quantization and sparsity on GPUs and propose a framework for structure pruning, to enable compressed deployment on a large variety of massively-parallel accelerators.

Together with considering design principles of DNNs, a methodology is introduced for targeting efficient deployment for virtually any modern hardware/software stack for DNNs. Several design principles for DNNs are discovered using this methodology, enabling the design of more efficient models without explicit compression.

Zusammenfassung

Tiefe Neuronale Netzwerke (DNNs) sind heutzutage eine Schlüsseltechnologie und der Haupttreiber für viele der jüngsten Fortschritte bei Anwendungen der künstlichen Intelligenz (KI), einschließlich Computer Vision, Verarbeitung natürlicher Sprache und Spracherkennung. DNNs zeigen die Fähigkeit, Trainingsdaten hervorragend anzupassen und gleichzeitig gut auf ungesehenen Daten zu verallgemeinern. Dies ist besonders effektiv, wenn große Datenmengen und ausreichende Hardwareressourcen verfügbar sind. Die Hardwareanforderungen im Bezug auf Berechnungen und Speicher sind der limitierende Faktor für ihren Einsatz in mobilen Systemen wie Handheld- oder Head-Word-Geräten.

Der Einsatz von DNNs in mobilen Geräten ist eine der wichtigsten Herausforderungen für die nächste Generation von KI-Anwendungen, einschließlich erweiterter Realität oder verbesserter Interaktionen zwischen Mensch und Computer. Für einen effektiven Einsatz müssen drei wichtige Forschungsrichtungen gemeinsam berücksichtigt werden: effizientes Modelldesign, Hochleistungshardware sowie kooperierende Software-Frameworks. Diese Arbeit untersucht diese Forschungsrichtungen unter einem ganzheitlichen Gesichtspunkt und berücksichtigt sorgfältig die Auswirkungen einer Richtung auf die anderen, um Techniken zu entwickeln, die den Gesamteinsatz verbessern.

Zunächst wird ein effizientes Modelldesign durch Komprimierung in Form von Quantisierung untersucht, um die erforderliche Datendarstellung von Gleitkommazahlen mit einfacher Genauigkeit auf Niedrigbitformate zu reduzieren. Es werden verschiedene Quantisierungstechniken evaluiert und eine Bibliothek eingeführt, die beliebige Bitkombinationen auf Hauptprozessor (CPUs) ermöglicht. Das Potenzial und die Auswirkungen der Abbildung quantisierter DNNs werden auf mobilen CPUs sowie Grafikprozessoren (GPUs) und rekonfigurierbare Prozessoren (FPGAs) eingehend untersucht.

Das nächste Kapitel befasst sich mit den Einschränkungen quantisierter DNNs und schlägt ein Co-Design für Komprimierung und Algorithmus vor, das auf einen Einsatz mit geringer Latenz auf mobilen CPUs abzielt und gleichzeitig eine hohe Vorhersagequalität erzielt. Der vorgeschlagene Komprimierungsalgorithmus basiert auf adaptiven Quantisierungsfunktion, die zusätzlich Sparsity in das DNN induziert. Ein Algorithmus wird eingeführt, um Berechnungen zu beschleunigen, indem die durch die Komprimierungstechnik erstellten Datenformate mit

aggressiv niedrigen Datentypen und dünn besetzten Matrizen ausgenutzt werden.

Die letzten Kapitel befassen sich mit den Nachteilen extremer Formen der Quantisierung und dünn besetzten Matrizen auf GPUs und schlagen einen Rahmen für die Reduktion von Strukturen vor, um eine komprimierte Bereitstellung auf einer Vielzahl von massiv parallelen Beschleunigern zu ermöglichen. Unter der Berücksichtigung der Entwurfsprinzipien von DNNs wird eine Methodik eingeführt, mit der eine effiziente Bereitstellung für praktisch jede moderne Hardware- / Software-Infrastruktur für DNNs angestrebt werden kann. Mit dieser Methode werden verschiedene Entwurfsprinzipien für DNNs entdeckt, die den Entwurf effizienterer Modelle ohne explizite Komprimierung ermöglichen.

Table of contents

1 Introduction **1**

2 Background **7**

2.1 Deep Neural Networks 7
2.2 Datasets 13
2.3 Hardware for Deep Learning 14
2.4 Software for Deep Learning 18

3 Resource-Efficient Neural Networks **21**

3.1 Quantized Neural Networks 23
3.2 Pruning Networks 26
3.3 Architecture Design 29

4 Quantized Inference **35**

4.1 Low-Precision Signed-Integer GEMM 35
4.2 QNNs on CPUs 46
4.3 QNNs on FPGA 50
4.4 QNNs on GPU 52
4.5 Resource Efficient Deep Eigenvector Beamforming 53
4.6 Summary 54

5 Reduce-and-Scale **57**

5.1 Quantization 58
5.2 Inference 60
5.3 Compression 64
5.4 Evaluation 66
5.5 N-Ary Quantization 71
5.6 Summary 82

6 Parameterized Structured Pruning **85**
6.1 Parameterization 86
6.2 Regularization 87
6.3 Pruning 89
6.4 Hardware-Friendly Structures 90
6.5 Experiments 93
6.6 Summary 99

7 Architecture Search **101**
7.1 Design Space Exploration 102
7.2 Evaluating the Efficiency of Building Blocks through Camuy . . . 107
7.3 Sigmoidal Residuals 108
7.4 Structure definitions 112
7.5 Elastic-net regularization 116
7.6 Evaluation 117
7.7 Summary 127

8 Comparing Compression Techniques on Hardware **129**

9 Discussion **133**
9.1 Potential and Limitations of Compression 133
9.2 Transferability of Insights 135
9.3 Broader Impact and Future Directions 138

10 Conclusion **141**

References **147**

Chapter 1

Introduction

Machine Learning (ML) is an emerging key technology and the main contributing factor for many recent performance boost in Artificial Intelligence (AI) tasks, including computer vision, natural language processing and speech recognition. Especially Deep Neural Networks (DNNs) are the currently predominant ML models, which exhibit the ability to excellently fit training data while also generalizing well to unseen data. While being the driving factor behind many recent success stories of AI, DNNs are notoriously data and resource hungry, a property which makes development and deployment difficult. Recent research showed that training can be scaled up to thousands of accelerators operating in parallel, resulting in a development phase not exceeding a couple of minutes, even for large-scale image classification. However, the deployment has usually much harder constraints than the development, as energy, space and monetary resources are scarce in mobile devices.

There exits an increasing variety of potential deep learning application where on-device inference is required and cloud access is not a viable solution. This is mainly due to latency reasons of interactive applications, but security or privacy considerations are also important factors in this context. For instance, Augmented Reality (AR) systems heavily rely on image classification, object detection and segmentation techniques, where DNNs usually demonstrate state-of-the-art performance. Such AR systems need to be deployed on a range of handheld and head-worn devices, which contain heterogeneous (i.e. CPU and GPU) and resource-constrained embedded systems. Another field of applications are signal enhancement or detection to implement noise cancellation or voice activation systems. These systems can potentially be implemented in in-ear

headphones using tiny processors, where latency and energy constraints are the crucial factor whether an application is viable. Automatic speech recognition systems are also highly desirable to run locally on the device in order to enhance interactions between humans and computers.

Three major research directions are necessary in order to effectively enable such applications: i) **Efficient models** aim to maximize prediction quality while minimizing hardware and software requirements. Research of this directions focuses on ML techniques for better structural efficiency where, for instance, architecture design principles are optimized or training enhancements are developed. Model compression through quantization (reducing the representation of operands) or pruning (inducing sparsity) are further ML optimizations for inference improvements. ii) **Efficient hardware** aims to maximize the yield of semiconductor technology for ML models. The focus here is on advanced processor technologies targeting either general-purpose or specialized use, with respect to application and energy demands. General-purpose technologies for instance, such as CPUs target high frequency and low off-chip accesses through large caches or GPUs trade frequency with massive-amount of parallelism for increased throughput. Domain-specific processors focus on a special use case or FPGAs feature configurable logic blocks which can be tailored to certain requirements. iii) **Efficient software** aims to maximize utilization and productivity when deploying ML models on hardware. One research directions targets high-level optimization with a focus on graph optimization or distribution algorithms for ML models. Another research direction develops compiler and code generation tools for a wide range of processors, while others focus on highly optimized solutions for a specific processor. These research directions need to be tightly coupled for efficient interactions and, hence, have to be developed with attention to each other.

This work studies these three research directions from an holistic point of view: it evaluates how redundancies of over-parameterized DNNs can be removed through compression and their respective potential and limitations when mapped onto hardware. The compression techniques exhibit here are quantization, pruning and architecture search, which are specifically designed and tailored to the targeted software/hardware platform. Mobile processors are used for developing the inference concepts and evaluation, including state-of-the-art CPUs, GPUs and FPGAs as well as their respective software stack. Based

on the techniques developed within this work in combination with extensive evaluation, it is examined which compression maps well to a certain processor and, ultimately, which compression/hardware codesigns are the most promising candidates.

Contributions

This work makes the following contributions:

- *Understanding quantized inference:* a detailed analysis of various data representations for DNNs is performed, ranging from half-precision floating point to extreme binarization forms. Model accuracy and inference performance is evaluated for several tasks, quantization techniques as well as hardware platforms, in order to elaborate on the effectiveness of quantized inference. This contribution comprises three publications [1]–[3].
- *Reduce-and-Scale (RaS):* a novel inference concept with a focus on low-latency and low-storage models on general-purpose CPUs. The concept leverages extreme forms of quantization and sparsity in combination with an efficient inference algorithm and without the need for specialized hardware. A novel n-ary quantization routine enables extreme data compression without accuracy degradation, even for complex tasks. This contribution includes two publications [4], [5].
- *Parameterized Structure Pruning (PSP) and architecture search:* a highly flexible framework for creating structured sparsity in DNNs without sacrificing prediction performance. The framework mainly targets DNN compression for massively-parallel accelerators and also enables architecture search which is compatible with the compiler and library stacks for utmost inference efficiency. Various evaluations on different tasks gained insights of design principles that can be used without explicit compression. This contribution includes two publications [6], [7].
- *Understanding compression and hardware for DNNs:* there is an extensive and ongoing discussion in the ML as well as computer architecture community on the most efficient compression and deep learning techniques as well as hardware accelerators. The overarching contribution of this work

is the understanding of compression on hardware from a perspective that covers both, ML and computer architecture research. While all chapters and publications within this work are required to generate the necessary insights, an overall analysis exhibiting potential and limitations is given in Chapter 8.

Organization of this Work

Chapter 2 provides the background on DNNs and the respective datasets used throughout this work. This chapter also gives a brief introduction into various hardware platforms as well as software frameworks for deep learning.

Chapter 3 gives an overview of current research directions in the field of resource-efficient DNNs, which build the foundation of this work. Compression techniques such as quantization and pruning as well as other forms of deep learning techniques are discussed that exhibit potential for memory and compute efficiency.

Quantization is extensively evaluated in Chapter 4 from various different views: first, a software library for variable bit inference is proposed for CPUs. Second, several quantization functions are reproduced to study the impact of efficient data representations. Then this library and other frameworks are leveraged in order to show the potential and limitations of quantized inference on CPUs, GPUs and FPGAs.

Chapter 5 uses the insights gained from Chapter 4 and proposes a novel inference technique denoted as Reduce-and-Scale (RaS). This technique leverages sparse and quantization models in combination with a novel computing algorithm to accelerate inference on general-purpose processors.

After analyzing the limitations of RaS on massively-parallel accelerators, Chapter 6 reviews structured sparsity patterns and proposes Parameterized Structured Pruning (PSP). This novel pruning technique enables variable structure definitions within DNNs and allows for hardware-defined computations.

Chapter 7 uses PSP and introduces a generic framework for architecture search. The proposed framework is able to find efficient sub-models in a larger architecture while being compatible with the software stack of state-of-the-art accelerators. Last, the framework is applied to identify general design principles,

which can be used to build efficient training and inference models without the pruning step.

An overall comparison of the various techniques is presented in Chapter 8, in order to evaluate the most promising compression technique and hardware platform. This chapter selects the best performing compression technique for a certain processor and compares the absolute inference performance for mobile CPUs, GPUs and FPGAs.

The insights with respect to potential and limitations as well as transferability of the results are discussed in Chapter 9 before Chapter 10 concludes this work.

Chapter 2

Background

This chapter provides the required background on deep learning techniques as well as hardware and software for implementing and realizing it. Each section briefly introduces the most significant concepts of the respective technology that is required to be able to follow this work. An in-depth introduction is out of scope of this work, because each topic represents its own field of research and the reader may use the provided references as further reading.

2.1 Deep Neural Networks

2.1.1 Feed-Forward Neural Networks

DNNs are constructed by layers of stacked processing units of linear transformations and non-linear activation functions. For each layer L, a unit computes an activation function of the form

$$\mathbf{x}^l = \phi(\mathbf{W}^l \oplus \mathbf{x}^{l-1} + \mathbf{b}^l), \tag{2.1}$$

where $\mathbf{W}^l$ is a weight tensor, $\mathbf{b}^l$ is a bias vector, $\mathbf{x}^{l-1}$ is an input tensor, $\oplus$ denotes a linear operation and ϕ is a non-linear activation function.

The most common non-linearities are the Rectified Linear Unit (ReLU) function $\phi(x) = \max(x, 0)$, the sigmoid function $\phi(x) = 1/(1+e^{-x})$, and the hyperbolic tangent function $\phi(x) = (e^x - e^{-x})/(e^x + e^{-x})$. Sigmoid and hyperbolic tangent function usually train much slower for deep architectures, due to the vanishing gradient problem and, hence, are not frequently used in DNNs. The

most prevalent non-linear operation is the ReLU function, as it features fast training speed by avoiding vanishing gradients. One drawback of the ReLU function are *dead neurons* that occur when certain neurons are never activated and can potentially require larger models. Variations of the ReLU functions that reduce dead neurons are Leaky ReLU, Parametric ReLU, (Scaled) Exponential Linear Unit, Gaussian Error Linear Unit and Swish. The main applications studied in this work are classification tasks, where the output layer is commonly activated by the softmax function as $\phi(\mathbf{x})_i = e^{x_i} / \sum_{j=1}^{K} e^{x_j}$, where K denotes the number of classes.

The two most common types of linear transformations $\oplus$ in feed-forward DNNs are fully-connected and convolution operations. The input to a fully-connected operator is a vector $\mathbf{x} \in \mathbb{R}^n$, where individual dimensions are denoted as *neurons*. Transformation of a fully-connected operation is implemented as a matrix-vector multiplication $\mathbf{W}\mathbf{x}$ where $\mathbf{W} \in \mathbb{R}^{m \times n}$ is a trainable weight tensor. Convolution operators are usually applied for data that exhibits spatial or temporal dimensions such as images or spectrograms. For instance, two-dimensional colour images can be represented as three-dimensional tensors $\mathbf{x}^l \in \mathbb{R}^{C \times W \times H}$, where C refers to the number of *channels* (i.e. red, green and blue) and W and H refer to width and height of the image, respectively. In this case, the convolution operator implements a four-dimensional weight tensor $\mathbf{W} \in \mathbb{R}^{K_w \times K_h \times C \times D}$, which produces an image $\mathbf{x}^{l+1} \in \mathbb{R}^{D \times W \times H}$ with d channels by computing

$$x_{d,w,h}^{l+1} = \sum_{k_w=1}^{K_w} \sum_{k_h=1}^{K_h} \sum_{c=1}^{C} W_{k_w,k_h,c,d} \cdot x_{c,i_w(w,k_w),i_h(h,k_h)}, \tag{2.2}$$

with i_w and i_h being indexing functions

$$i_w(w,k_w) = w - \left\lfloor \frac{K_w}{2} \right\rfloor - 1 + k_w \qquad \text{and} \qquad i_h(h,k_h) = h - \left\lfloor \frac{K_h}{2} \right\rfloor - 1 + k_h. \tag{2.3}$$

The four-dimensional weight tensor $\mathbf{W}$ can be seen as a set of D *filter kernels*, each of spatial filter size $K_w \times K_h$. Each spatial location of a particular output channel d in an intermediate image $\mathbf{x}_d^l$ is computed from a $K_w \times K_h$ region of the input image $\mathbf{x}^{l-1}$ and the same filter is shifted over the spatial dimensions of the feature map. This reusing of the filter leads to a detection of features within the input images $\mathbf{x}^{l-1}$ that is invariant with respect to its spatial location. In

the context of CNNs, individual channels of intermediate images $\mathbf{x}_d^l$ with $l > 0$ are also referred to as *feature maps*.

For data exhibiting spatial or temporal dimensions, it is also desired to have a feature detection mechanism that is invariant with respect to scale. Therefore, CNNs typically employ a *pooling* operation that merges spatially neighbouring values within a feature map to reduce the feature map's size. Common choices are max-pooling and average-pooling which combine the results of neighbouring values by computing their maximum or average, respectively. Note that to some extend a scale invariant feature detector is also obtained by stacking multiple convolution layers, since the number of features influencing a particular spatial location increases with each layer. For instance, using two convolution layers, each having 3×3 filter kernels, results in each output spatial location being influenced by a 5×5 region from the input feature maps. This 5×5 region is also called the *receptive field*.

2.1.2 Training

Training neural networks aims to adjust the randomly initialized weight tensors $\mathbf{W}$ in a way that they solve a given task, for instance predicting correct classes for unseen inputs $\mathbf{x}$. This can be done by minimizing a loss function $\mathcal{L}$ using gradient based optimization. For supervised learning with a labelled training data set $\mathcal{D} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N)\}$ containing N input-target pairs, a typical loss function has the form

$$\mathcal{L}(\mathbf{W}; \mathcal{D}) = \sum_{n=1}^{N} l(y(\mathbf{W}, \mathbf{x}_n^0), t_n) + \lambda r(\mathbf{W}), \tag{2.4}$$

where $l(y_n, t_n)$ is the *data term* that penalizes the DNN parameters $\mathbf{W}$ if the output y_n does not match the target value t_n. $r(\mathbf{W})$ is a *regularizer* that prevents the DNN from overfitting, which is commonly the ℓ_1 or ℓ_2 term, and $\lambda > 0$ is a hyper parameter for adjusting the regularization strength. For classification, the loss function is usually implemented using the cross entropy while for regression squared error loss is used. Backpropagation computes the gradient for a fixed input–target pair, where the training process minimizes the loss function by

iteratively computing

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{W}; \mathcal{D}), \tag{2.5}$$

where η is a tunable learning rate parameter and t is the current iteration. Simply computing the update as in (2.5) is not possible for most cases, because the size of the dataset N is usually large and the gradient $\nabla_{\mathbf{W}} \mathcal{L}$ requires too much memory, since the sum over all N samples is computed. This can be resolved by approximately computing only a subset (or *mini-batch*) of N_B randomly selected samples from the dataset, which is denoted as *Stochastic Gradient Descent (SGD)* [8]. SGD is the simplest optimizer commonly used for training DNNs but has a large variety of similar variants. This work uses mainly the popular Momentum optimizer, which filters out fluctuations between iterations and ultimately accelerates training.

2.1.3 Normalization

Scaling neural networks in depth rather than width is usually more effective in order to increase prediction performance for a given compute and memory complexity. This scaling, however, also increases the difficulty of gradient-based optimization, because vanishing or exploding gradients are influencing training negatively. Vanishing and exploding gradients can occur when each layer shrinks its input by a factor $\alpha < 0$ and $\alpha > 1$ so that the outputs of the lth layer will be exponentially scaled by α^l. Therefore, most modern DNN architectures employ normalization layers after the linear transformation. The most prevalent normalization technique for convolution layers is batch normalization [9] which computes

$$\mu_{\mathbf{x}_d^l} \leftarrow \frac{1}{N_B} \sum_{n=1}^{N_B} x_d^l, \qquad \sigma^2_{\mathbf{x}_d^l} \leftarrow \frac{1}{N_B - 1} \sum_{n=1}^{N_B} (x_d^l - \mu_{\mathbf{x}_d^l})^2, \qquad \mathbf{x}_d^l \leftarrow \frac{\mathbf{x}_d^l - \mu_{\mathbf{x}_d^l}}{\sigma_{\mathbf{x}_d^l}} \cdot \gamma_d + \beta_d, \tag{2.6}$$

where β_d and γ_d are new trainable parameters for the D feature maps. Batch normalization aims to rescale the activation statistics in each feature map to zero mean and unit variance to eliminate the effect of exponential up- or downscaling. The additional trainable scale parameters γ_d and shift parameters β_d are intro-

duced to recover the ability of the DNN to approximate any desired function. While batch normalization is nowadays essential for fast convergence, it also adds statistical noise to the activations which usually achieves better generalization performance.

2.1.4 Neural Architectures

Most architectures for classification tasks follow a rather simple technique of stacking layers and building blocks. However, during the evolution of architectures several additional components and extensions have been proposed that resulted in better performance. This section gives an overview of the most popular DNN architecture which are also heavily used in this work.

AlexNet

The AlexNet architecture [10] was the first work to show that DNNs are capable of improving over conventional hand crafted computer vision techniques by achieving around 16.4% Top-5 error on the ILSVRC12 challenge – an improvement of approximately 10% absolute error compared to the second best competitor in the challenge who relied on well established computer vision techniques. The architecture consists of eight layers — five convolution followed by three fully connected layers. This most influential work essentially started the advent of DNNs, which can be seen from the fact that DNNs have spread over virtually any scientific field dealing with data and achieving improved performances over well established methods in the respective fields. AlexNet was designed such that training could be performed efficiently on two GPUs rather than following some clear design principle. This involves the choice of heterogeneous window sizes $K_w \times K_h$ and seemingly arbitrary numbers of channels per layer C. Furthermore, convolutions are performed in two parallel paths to facilitate the training on two GPUs.

VGGNet

The VGGNet architecture [11] won the second place at the ILSVRC14 challenge with about 7.3% Top-5 error. Compared to AlexNet, its structure is more uniform and with up to 19 layers much deeper. Its design is guided by two main principles. (i) VGGNet uses mostly 3×3 convolutions and increases its receptive field by

stacking several of them. (ii) After downscaling the spatial dimension with 2×2 max-pooling, the number of channels should be doubled to avoid information loss. From a hardware perspective, VGGNet is often preferred over other architectures because of its uniform architecture.

InceptionNet

The winner of the ILSVRC14 challenge was with 22 layers an even deeper architecture named InceptionNet or GoogLeNet [12] with 6.7% Top-5 error. The main feature of this architecture is the inception module, which combines the outputs of convolutions with the different filter sizes 1×1, 3×3, and 5×5 by stacking them. To reduce the computational burden, InceptionNet adapts the use of 1×1 convolutions proposed in [13] immediately before the larger 3×3 and 5×5 convolution to reduce the number of channels and, therefore, decrease the computational burden.

ResNet

Motivated by the observation that adding more layers to conventional architectures does not necessarily reduce the training error, residual networks (ResNets) introduced by He, Zhang, Ren, *et al.* [14] follow a rather different principle. The key idea is that every layer computes a residual that is added to the layer's input which is often graphically depicted as skip connections. The authors hypothesize that identity mappings play an important role and that it is easier to model them in ResNets by simply setting all the weights to zero instead of simulating an identity mapping by adapting the weights of several consecutive in an intertwined way. In any case, the skip connections reduce the vanishing-gradient problem during training and enable extremely deep architectures of up to 152 layers on ImageNet or even up to 1000 layers on CIFAR-10. With this technique, ResNet won the ILSVRC15 challenge with 3.6% Top-5 error.

DenseNet

Inspired by ResNets whose skip connections have shown to reduce the vanishing gradient problem, densely connected DNNs (DenseNets) introduced by Huang, Liu, Maaten, *et al.* [15] drive this idea even further by connecting each layer to all previous layers. Instead of adding the output of a layer to its input, DenseNets

stack the output and input of each layer. Since this stacking necessarily results in an increasing number of feature maps with each layer, the number of new feature maps computed by each layer is relatively small and it is proposed to use compression layers after downscaling the spatial dimension with pooling. In this way, DenseNets allow for even deeper architectures and they are more parameter and computation efficient than ResNet, respectively. However, this architecture is highly non-uniform which complicates the hardware mapping and ultimately slows down training.

2.1.5 Straight-Through Gradient Estimator

Many recently developed methods for resource-efficiency in DNNs incorporate a component in the computation graph of the loss function $\mathcal{L}$ that is non-differentiable or has zero gradient almost everywhere, which prevent the use of conventional gradient-based optimization. The Straight Through Gradient Estimator (STE) [16] is a simple but effective way to approximate the gradient of such components by simply replacing their gradient with a non-zero value. Let f be some non-differentiable operation within the computation graph and w be its input such that the partial derivative $\partial\mathcal{L}/\partial w$ is not defined. The STE then approximates the gradient $\partial\mathcal{L}/\partial w$ by

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial f}\frac{\partial f}{\partial w} \approx \frac{\partial \mathcal{L}}{\partial f}\tilde{f}'(w), \tag{2.7}$$

where $\tilde{f}'(w)$ is an arbitrary non-zero surrogate derivative. The simplest choice $\tilde{f}'(w) = 1$ which simply passes the gradient on to higher components in the computational graph. More involved strategies select $\tilde{f}'$ to be the derivative of some differentiable function $\tilde{f}$ that has a similar shape as the non-differentiable function f, e.g., for $f = \mathrm{sign}$ one could select $\tilde{f}' = \tanh'$.

2.2 Datasets

In order to evaluate the effectiveness and generalization of certain deep learning techniques and software/hardware infrastructure, it is necessary to use reasonable data. First, data needs to represent real-world examples where the respective technique is of interest. Second, the chosen datasets require to be a community

default in order to make different techniques comparable. This section briefly introduces the various datasets and tasks used to evaluate performance metrics of compression techniques on hardware in this work.

MNIST dataset is a collection of 28x28 images of handwritten digits with 60 thousand training and 10 thousand testing images. This dataset features 10 classes for each number in the range of [0,9].

Street View House Numbers (SVHN) dataset [17] is similar to MNIST but features significantly harder real-world images of digits house numbers taken from Google Street View. The dataset contains 600 thousand labelled images of 32x32 resolution featuring 10 classes, one for each digit.

CIFAR datasets [18] contain low-resolution (32x32) real-world images that allow quick evaluation of image-recognition systems. CIFAR-10 features 10 classes, such as airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, with 6000 training and 1000 testing images per class. CIFAR-100 is very similar but feature 100 classes that are grouped into 20 superclasses with 500 training and 100 testing images per class.

Large Scale Visual Recognition Challenge 2012 (ILSVRC2012 or ImageNet) dataset [19] consists of 10 million large hand-labeled images using 10 thousand categories. Each image has a list of at most 5 object categories in the descending order of confidence, where the tasks is to identify multiple objects (i.e. Top1 and Top5 accuracy) in an image. The ImageNet dataset is considerably harder than previous mentioned tasks and is one of the most challenging tasks in the context of resource efficiency, since accurate models require extreme compute and memory resources.

2.3 Hardware for Deep Learning

Improvements in hardware for deep learning are the key driver for the recent success story of AI applications through DNNs. Both, training and inference have extreme high demands on their targeted platform and certain hardware requirements can be the deciding factor whether an application is possible. This section briefly introduces the most important hardware for deep learning and discusses their potential as well as limitations. While this discussion is generic and independent to training or inference, it should be noted that all processor concepts are available in different scales, ranging form mobile to server variants.

2.3.1 CPUs

CPUs are originally designed to optimize single-thread performance, in order to execute an individual computation within the shortest possible latency. Unfortunately, single-thread performance is stagnating since the end of Dennard scaling [20], and now performance scaling usually requires parallelization. While multithreading is a rather obvious solution for parallelization that is applicable to many tasks, vectorization is a technique that promises great potential for certain applications. Vectorization leverages the Single-Instruction Multiple-Data (SIMD) paradigm and exploits the low cost of data-level parallelism in current CMOS processes. CPUs show excellent properties of exploiting sparse neural networks due to their short vector units and the low amount of multithreading together with high frequency. Furthermore, they usually support 8-bit integer formats and feature certain instructions for extreme low representation and are consequently well suited for quantization operations.

2.3.2 GPUs

GPUs are initially designed to accelerate image and video processing only and are nowadays the most popular general-purpose accelerators for a many tasks, such as scientific and AI computations. The architecture consist of many streaming multiprocessors which are highly parallel and each implements many light-weight cores. Thus, GPUs are massively-parallel processors with large memory that provides extremely high bandwidth and throughput, but significantly lower frequency in comparison to CPUs. The extreme high amount of parallelism and the resulting demand on structured computations, however, virtually prevents the deployment of sparse computations. Modern GPU designs and their respective software stack implement support for reduced-precision computations, such as 8-bit integer and half-precision floating point formats, which are very well suited for deep learning. More extreme forms of quantization are not yet support and do not result into more efficient inference or training.

2.3.3 FPGAs

Field Programmable Gate Arrays (FPGAs) are a family of processors that implement a large array of configurable logic blocks which can be programmed

using hardware-description languages (e.g. VHDL, Verilog, HLS). This concept is the main difference to ASICs in terms of technology, since hardware can be designed for specific application or functional requirements. While this reconfigurability enables various opportunities that go beyond the capabilities of CPUs and GPUs, it comes at the cost of much lower frequency and reduced on-chip memory. FPGAs are in principle very well suited for neural networks, since compute units can be specifically tailored to fit the diverse computations while also enabling massive amounts of parallelism. Reconfigurable hardware is especially interesting for compressed neural networks due to their flexibility to implement any data formats as well as sparse logic.

2.3.4 Domain-Specific Accelerator

Recent interest in deep learning has motivated to push advancements in the development of custom accelerators, such as Google's TPUs and Graphcore's IPU. The key feature of the TPU (and most of other deep learning accelerators) is a 256x256 matrix-multiplication unit that is referred to as systolic array. Systolic arrays are a variant of massively-parallel processor arrays, very suitable for regular problems such as linear algebra operations, and a promising candidate to address the increasing costs of data movements. The objective of such arrays is to minimize instruction-fetch and data-access costs by constraining the data flow to matrix and vector operations. However, data movements can only be reduced if locality effects are sufficiently exploited and the data flow constraints of a systolic array may result in poor utilization and latency increase. Such domain-specific accelerators are usually very constraint when aiming to optimize neural network through compression. For instance, the TPU allows 8-bit integer and half-precision floating point formats while other (potentially lower-precision) representations are not efficiently supported by hardware. Furthermore, the dense structure of the systolic arrays demand for highly dense computations and can not exploit fine-grained sparsity patterns.

2.3.5 Loop-Back vs. Data-Flow Architectures

One can roughly categories hardware platforms for deep learning inference into loop-back and data-flow architectures: loop-back architectures use a fixed processor and memory system to move data from off-chip memory to the processor

and leverage the available compute resources. This is performed for each layer or operation sequentially until the inference is done. The drawback of loop-back architectures is that it potentially requires many data movements from and to off-chip memory, which is time and energy consuming. CPUs aim to reduce these memory accesses by featuring large on-chip caches and reuse data as much as possible. Similar are domain-specific accelerators, such as TPUs, which usually feature a large and programmable scratch pad memory on chip. On the contrary, GPUs feature large register files and aim to hide memory latency by leveraging parallel slackness. Another critical aspect of loop-back architectures is low compute utilization, which can potentially occur if certain layer or operation types do not fit the static compute array (i.e. if operation size is too low). The advantage of such generic compute architecture is, that they allow arbitrary operations in combinations with productive code generations, since the hardware does not need to be optimized for a certain task. Continuous improvements in semi-conductor and processor technology are the main improvement factor of such inference engines.

In contrast to this, data-flow architectures use a reconfigurable processor and memory system for computing the inference. Here, each layer or operation within a neural architecture is assigned a dedicated compute engine and its own memory subsystem, in order to enable inference in a pipelined fashion. This avoids off-chip accesses for intermediate operations completely by simple forwarding the computed results to the next hardware layer. Furthermore, data-flow architectures achieve excellent utilization of the available hardware logic, since several compute engines can be tailored to the required operation type and latency. One drawback of this inference architecture is, however, that it requires long development costs, because it does not only require software but also hardware optimizations. In addition, reconfigurable hardware comes at the cost of reduced absolute compute power in comparison to ASIC designs. The main limitation of data-flow architectures is that they demand the entire neural architecture (weights and activations) to stay on chip, which is highly restrictive for large models.

2.4 Software for Deep Learning

Software frameworks are especially important for deep learning applications because they are responsible for leveraging the available compute hardware most efficiently while also offering the user a productive working environment. Such software stacks can be roughly clustered into training frameworks for developing models and inference frameworks for deploying models. This section briefly introduces the most important software frameworks in the context of deep learning that are used throughout this work.

2.4.1 Training Frameworks

There exists a large variety of deep learning frameworks that enable training and inference of models on different hardware platforms. Such frameworks offer interfaces for expressing ML models and an implementation for executing the algorithms. Abstract expressions of an algorithm flow can be executed without major modifications on a wide variety of heterogeneous systems, ranging from mobile CPUs up to large-scale distributed systems containing hundreds of GPUs or TPUs.

Some of the most popular frameworks are Theano, TensorFlow, PyTorch, etc. which all map tensor operations onto hardware using highly optimized libraries or code generation. These frameworks make gradient based optimization particularly convenient: the user specifies the loss as a computation graph and the gradient is calculated automatically by the framework using the backpropagation algorithm. While most of such frameworks use the same libraries, they usually differ in terms of graph optimizations and degrees of freedom when implementing novel features.

2.4.2 Inference Frameworks

Training frameworks also support inference (or deployment) of models, however, they are optimized for the learning process and do not effectively take advantage of optimization techniques for deployment. On the contrary, specialized inference frameworks are able to exploit certain inference potentials, which allow faster inference, reduced memory footprint or less memory accesses. As a consequence, it is an obvious step to train models in one framework and export the trained

models into a more specialized framework, in order to achieve highest performance for deployment.

The TensorRT framework includes inference optimizer and runtime that delivers low latency and high throughput for deep learning inference on NVidia GPUs. It features different weight and activation formats in order to maximizes throughput by quantizing models and layer fusion techniques to optimizes use of GPU memory and bandwidth. Other features are auto tuning to selects best data layers and algorithms for a targeted GPU platform as well as dynamic tensor memory, which reduces the memory footprint and reuses memory for tensors efficiently.

Another deployment framework is TensorFlow-Lite, which is similar to TensorRT but not vendor specific, thus, allows efficient inference on CPUs, GPUs and TPUs. TensorFlow-Lite also exploits model compression through quantization, specialized formats for reduced memory consumption as well as layer fusion techniques. One of the key features of this frameworks is a focus on the use of platform APIs for accelerated inference on various devices and small binary files.

2.4.3 Code and Hardware Generation

Inference frameworks, such as TensorRT and TensorFlow-Lite, are developed for a certain use case or hardware platform. They heavily rely on pre-defined layers and operations, which are ultimately mapped onto hardware platforms through specialized libraries (i.e. cuDNN or GemmLowp). However, these frameworks quickly fail to support deployment if novel operations or hardware is targeted.

TVM [21] is a compiler which exploits graph-level and operator-level optimizations for high-performance deployment across diverse hardware backends (i.e. ASIC, FPGA). The framework also supports low-precision formats, layer fusion as well as techniques for reducing the memory footprint. The main difference to other inference frameworks is that it automates optimization of low-level routines for hardware by employing techniques for rapid exploration of code optimizations. This makes the framework versatile to new compute platforms as well as model features.

FINN [22] is a inference framework which is specialized in hardware generation for Xilinx FPGA. It exploits extreme forms of quantization in order to generate on-chip data-flow architectures, customized for each model. The resulting inference

accelerators are extremely efficient in terms of throughput, latency and energy. However, FINN is an vendor dependent experimental framework, which relies on trained quantization and does not support various operation and layer types.

Chapter 3

Resource-Efficient Neural Networks

DNNs prove particularly effective when big amounts of data and ample computing resources are available. In real-world applications, however, the resources during inference are typically limited, effectively preventing the resource-hungry models to be deployed. Figure 3.1 illustrates three key challenges which have to be jointly considered to facilitate DNNs in real-world applications:

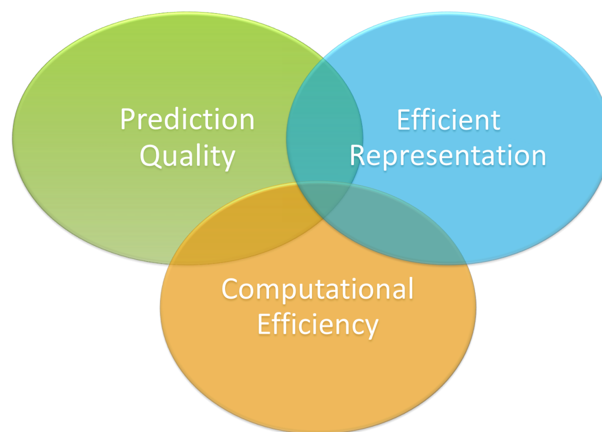


Fig. 3.1 Three key aspects of resource-efficient neural networks. [2]

i) *Efficient Representation*: model requirements in terms of memory (i.e. parameters and activations) should match the usually limited resources in deployed systems. ii) *Computational Efficiency*: the model should be computationally efficient during inference, utilizing the available hardware optimally with respect to time and energy. iii) *Prediction Quality*: model complexity versus prediction qual-

ity trade-offs must be considered to achieve good prediction performance while simultaneously reducing computational complexity and memory requirements.

These three aspect have motivated recent research interest in the field of resource-efficient techniques for deep learning. This chapter gives an extensive overview of the current research directions of such techniques, which are concerned with reducing the model size and/or improving inference efficiency while at the same time maintaining accuracy levels close to state-of-the-art models. The content of this chapter has been published in collaboration with Wolfgang Roth as an preprint [2].

There are three major directions of research concerned with enhancing resource-efficiency in DNNs, which are:

Quantized Neural Networks Weights and activations of neural networks are usually represented as single-precision floating point values and during inference billions of floating-point operations are carried out. Quantization techniques aim to reduce the number of bits used to represent these vaules, in order to reduce the memory footprint and area cost to facilitates faster inference using cheaper arithmetic operations.

Network Pruning Starting from a potentially large neural architecture, pruning techniques aim to remove parts of the model during or after training. The parts being removed range from individual parameters (denoted as unstructured pruning) to a more global scale of neurons, channels, or even entire layers (denoted as structured pruning). This structure granularity ultimately depends on the hardware and software infrastructure of the target computing platform.

Structural Efficiency This research direction includes a diverse set of techniques that achieve resource-efficiency at the structural level of neural networks. *Knowledge distillation* is a technique where a small student model is trained to mimic the behaviour of a larger teacher. The idea of *weight sharing* is to use a small set of weights that is shared among several connections of the neural network in order to reduce the parameter footprint. Several works have investigated *special matrix structures* that require less parameters and allow for faster matrix multiplications. Furthermore, there exist several *manually designed architectures* that introduced lightweight building blocks or modified existing ones to enhance resource-efficiency.

Most recently, *neural architecture search* has emerged that discover efficient neural architectures automatically.

These techniques are not mutually exclusive and they can potentially be combined to further enhance resource-efficiency. For instance, one can both sparsify a model and reduce arithmetic precision while also optimizing by apply structural efficiency techniques.

3.1 Quantized Neural Networks

Quantization aims to reduce the bit width for weights and/or activations of a DNN from single-precision floating point (real-valued) to more efficient formats. Reducing the number formats results in less storage requirements and in less memory accesses, which are the most energy and time consuming operations when computing predictions. Furthermore, low-bit formats can achieve higher throughput or lower latency, since area savings allow for more parallelism due to simpler instructions. The main challenge is to reduce the number of bits as much as possible while maintaining prediction quality close to that of a real-valued model. An extreme example of quantization are binary weights $w \in \{-1, 1\}$ together with binary activations $x \in \{-1, 1\}$, which enables inference computations through hardware-friendly logical bit operations. This section provides a literature overview of popular techniques that enable quantized neural networks.

3.1.1 Early Quantization Techniques

Low-precision computations within ML models date back at least to the early 1990s: Höhfeld and Fahlman [23], [24] rounded the weights during training to fixed-point formats with varying numbers of bits. This technique resulted in small gradient updates that are rounded to zero and, consequently, stalling training which they resolved by adding stochastic rounding. The same concept was recently successfully adapted to modern DNNs by Gupta, Agrawal, Gopalakrishnan, *et al.* [25].

Courbariaux, Bengio, and David [26] explored various numeric formats - such as float point, fixed point and dynamic fixed point - for DNNs using varying number of bits Avoiding multiplications through binary or ternary formats for

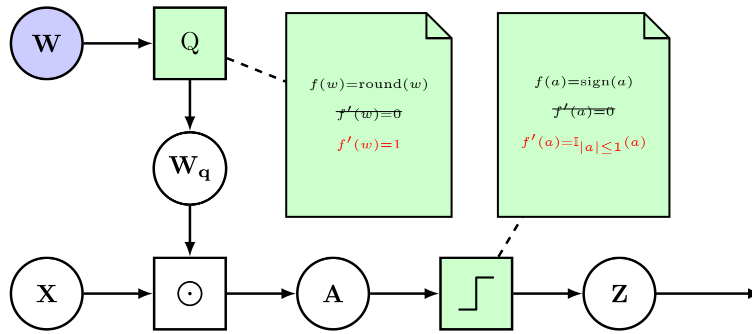


Fig. 3.2 A simplified building block of a neural network using the straight-through estimator (STE). [2]

weights by stochastic quantization was proposed by Lin, Courbariaux, Memisevic, *et al.* [27]. In Addition, activations are quantized to powers of two to enable cheaper bit-shift operations for inference. Lin, Talathi, and Annapureddy [28] use fixed-point formats for pre-trained real-valued models, where the required bit width is identified through an optimization formulation that considers the signal-to-quantization noise ratio.

3.1.2 Quantization-aware Training

Quantization routines usually deploy piece-wise constant functions with either undefined or zero gradients, which prevents their use in gradient-based learning optimizations. In recent years, the STE [16] became the most prevalent technique to approximate the gradients. In this context floating-point weights are usually maintained and quantized during forward propagation and, during backpropagation, the gradients are propagated through the quantization functions by assuming that their gradient equals one. The whole process is illustrated in Figure 3.2, where floating-point weights are updated using gradients computed at the quantized weights. At deployment, the full-precision weights are discarded and only the quantized low-precision weights are kept. This is denoted as quantization-aware training and it is the most popular way of reducing the number formats, as it achieves much better prediction quality than previous discussed techniques.

[29] quantized the weights of DNNs to a binary format where they consider deterministic and stochastic rounding during training (using the STE technique), which dramatically reduces storage requirements. A similar technique was

proposed by [30]: in addition to binary weights they also quantize activations to a binary format using the sign function and STE. This technique not only reduces storage requirements, but also activation memory and inference computations can be performed using hardware-friendly logic operations.

Li, Zhang, and Liu [31] trained ternary weights $w \in \{-a, 0, a\}$ by setting weights lower than a defined threshold Δ to zero, and setting weights to either $-a$ or a otherwise. Li, Zhang, and Liu [31] quantized weights to a ternary format $w \in \{-a, 0, a\}$ by thresholding the floating-point weights into three clusters, where they aim to minimize the ℓ^2 norm between floating point and ternary weight matrix. A different a asymmetric ternary format $w \in \{-a, 0, b\}$ was proposed by Zhu, Han, Mao, *et al.* [32] by leveraging trainable scaling $a > 0$ and $b > 0$ that are optimized using gradient descent as well as different threshold function.

Similar to the ternary quantization techniques, Rastegari, Ordonez, Redmon, *et al.* [33] use a floating-point scalar to scale the binary weights in each filter. Lin, Zhao, and Pan [34] quantize weights through linear combinations of multiple binary weight filters. Using the observation that weights and activations typically exhibit a non-uniform distribution, Miyashita, Lee, and Murmann [35] proposed to quantize values to powers of two. Cai, He, Sun, *et al.* [36] proposed a half-wave Gaussian quantization that approximates the predominant ReLU activation function better. Incremental network quantization [37] partitions weights of a DNN into sets, where each set is quantized sequentially during a retraining step until all sets are quantized. Jacob, Kligys, Chen, *et al.* [38] simulate the quantization step during training in order to reduce the format to 8-bit integer for deploying the models on general-purpose integer hardware. Liu, Wu, Luo, *et al.* [39] use the observation that shortcut connections of ResNet architectures are very sensitive to extreme forms of quantization. By only binarizing the residual path and leaving the shortcut connections to floating point, they significantly improve accuracy while achieving the potential of fast binary convolutions through bit-wise operations.

One of the most accurate quantization techniques was proposed by Zhang, Yang, Ye, *et al.* [40], where a learnable quantizer is used that reduces the quantization error. They use the fact, that fixed-point numbers can be seen as linear combination $\mathbf{v}^T\mathbf{b}$ with $\mathbf{v} = \{2^0, \ldots, 2^K\}$ and $\mathbf{b} \in \{0,1\}^K$ and view $\mathbf{v} \in \mathbb{R}^K$ as trainable parameters. This technique can be used for arbitrary bit representations, however, training time increases gradually with the targeted

amount of bits.

Zhou, Ni, Zhou, *et al.* [41] proposed a framework that allows for arbitrary bit combinations for weights, activations as well as gradients, in order to additionally leverage low-precision formats for training. In [42], weights, activations, weight gradients, and activation gradients are subject to customized quantization functions that allow for variable bit widths and facilitate integer arithmetic during training and testing. In contrast to [41], the work in [42] accumulates weight changes to low-precision weights instead of full-precision weights.

Another emerging trend towards reducing bit formats are techniques that target mixed-precision quantization, where different layers obtain different bit representations. Finding well approximating bit combinations is difficult due to the large design space. Hardware-aware automatic quantization [43] supports mixed precision formats (1-8 bits) where they find the optimal bit width for each layer by considering a particular hardware architecture. The framework leverages reinforcement learning to automatically determine the quantization policy and it takes the hardware accelerator's feedback in the design loop. Uhlich, Mauch, Cardinaux, *et al.* [44] propose to parameterize the quantizer with the step size and dynamic range where the bit width can then be inferred from them. They show that a suited parameterization is the key to achieve a stable training and a good final performance. Cai, Yao, Dong, *et al.* [45] proposed a framework for mixed-precision quantization without any access to the training or validation data. They optimize for a distilled Dataset, which is engineered to match the statistics of batch normalization across different layers of the network. An automatically determination of the mixed-precision bit setting for all layers is used by a Pareto frontier based on a sensitive evaluation.

While most work on quantization based approaches is empirical with a focus on prediction performance, some recent work gained more theoretical insights [46], [47].

3.2 Pruning Networks

Pruning techniques aim to compress DNNs using sparsity by setting certain parameters to zero, in order to reduce the memory consumption and to speed up computations. Pruning variants can be roughly categorized into two cluster: unstructured and structure pruning. Unstructured pruning sets individual weights

of a neural network to zero, without considering their position in a weight tensor. Such forms are typically easier to implement and obtain a large amount of sparsity without reducing prediction quality, however, they are difficult to implement (or not viable) on parallel hardware. On the contrary, structured pruning techniques set more coarse-grained structures of a tensor to zero, e.g., a whole channel of a convolution tensor.

3.2.1 Unstructured Pruning

LeCun, Denker, and Solla [48] proposed *optimal brain damage*, an algorithm for adapting the size of a neural network. They remove unimportant weights from a model by using second-derivative information to trade-off training error and model complexity. Similar to this, Hassibi and Stork [49] proposed to use an approximated full covariance matrix instead to prune weights that cause the least increase in loss function.

These initial techniques for network pruning were designed using small neural models and are not viable for modern DNN, since second-derivative information as well as covariance matrix are too compute intensive for such large models. As a result, most modern pruning techniques rely on some simpler heuristics for evaluating the importance of weights. Han, Pool, Tran, *et al.* [50] apply magnitude-based threshold pruning by iterating between pruning and re-training the model. Despite its simplicity, this technique is able to remove impressive amounts of weights and is one of the most adopted pruning variants nowadays. It is also integrated into *deep compression* [51], a technique for combined pruning and quantization, which also leverages Huffman coding for further compression.

Guo, Yao, and Chen [52] proposed a dynamic pruning technique, which allows wrongly pruned weights to be reincorporated again. Their techniques also relies on threshold-based pruning but uses additional auxiliary weights to reappear if their value exceeds a certain threshold.

3.2.2 Structured Pruning

Wen, Wu, Wang, *et al.* [53] use group lasso regularization in order to enforce sparsity induction in a structured way dynamically during training. They control the pruning amount simply by tuning the regularization strength and evaluate several interesting structures, such as channel or entire layers. Another

regularization-based technique was proposed by Liu, Li, Shen, *et al.* [54], where the γ parameters of batch normalization are used. They add an ℓ^1 regularization term to the loss function, which punishes non-zero γ parameters and, ultimately, forces certain parameters to zero. This results into pruning of whole channels, since each γ parameter is shared across a feature map. Huang and Wang [55] proposed trainable scaling factor to scale the outputs of specific structures, such as neurons, groups or residual blocks. Again, these parameters are regularized using an ℓ^1 penalty and certain parameters - and consequently structures - are forced to zero. Gordon, Eban, Nachum, *et al.* [56] proposed MorphNet, a technique to automate the design of DNN structures. They iteratively shrink and expand a network by regularizing structures with respect to computations or size. Thus, they not only enable pruning models, but also redesigning a baseline architecture dynamically to the demands of certain computation and memory requirements.

Luo, Wu, and Lin [57] proposed to evaluate the importance of certain structures by pruning channels that result in the least activation change in the subsequent layer. Louizos, Welling, and Kingma [58] propose to multiply weights with stochastic binary 0-1 gates associated with trainable probability parameters that effectively determine whether a weight should be pruned or not. They formulate an expected loss with respect to the distribution over the stochastic binary gates, and by incorporating an expected ℓ^0-regularizer over the weights, the probability parameters are encouraged to be closer to zero. To enable the use of the re-parameterization trick, a continuous relaxation of the binary gates using a modified binary Gumbel softmax distribution is used [59]. They show that their approach can be used for structured sparsity by associating the stochastic gates to entire structures such as channels. Li and Ji [60] extended this work by using the recently proposed unbiased ARM gradient estimator [61] instead of using the biased Gumbel softmax approximation. Aflalo, Noy, Lin, *et al.* [62] distill the knowledge from the over-parameterized parent network's inner layer by formulating the network pruning as a Knapsack Problem. This optimizes the trade-off between the importance of neurons and their associated computational cost and the pruned model is fine-tuned under the supervision of the parent network.

3.3 Architecture Design

Architecture design aims to find optimal neural architectures for a given task by evaluating design principles and building blocks. Several of these techniques have emerged over the past years, where the large majority targets resource-efficient neural architectures. Architecture design can be mainly clustered into *manual architecture design* and *neural architecture search*.

Manual Architecture Design

Initial DNN architectures for classification tasks are designed to use a set of convolution layers for feature extraction followed by one or more fully-connected layers for calculating the class probabilities. At the transition between convolution and fully-connected layers, feature maps are usually reshaped into vectors that are connected to the fully-connected layers. This transition is extremely parameter consuming because a large number of feature maps are used at this point. Lin, Chen, and Yan [13] proposed to use *global average pooling* to reduce the number of parameters at this transition, by reducing the spatial dimension of each feature map into a single feature through averaging. This does not only reduce the overall parameter requirements, but also results into better generalization by removing the spatial location of features before classification. They also leverage 1×1 convolutions with weight kernels $\mathbf{W} \in \mathbb{R}^{1 \times 1 \times C \times D}$, in order to reduce computation and parameter demands of the model. These 1×1 convolutions can be seen as feature pooling layers by performing the operation of a fully-connected layer over each spatial location across feature maps.

Many popular architectures [12], [14], [15] adopted these two techniques to scale model size while aiming for resource-efficient architectures. While global average pooling is used in virtually any modern DNN for classification, 1×1 convolutions are leveraged in various forms: Inception [12] splits standard $K \times K$ convolutions into a cheaper 1×1 convolution for reduce the number of feature maps before a subsequent $K \times K$ convolution is performed. ResNet architectures [14] use bottleneck structures in the residual path, where a 1×1 convolution is used for reducing the number of feature maps before the computational heavy 3×3 convolution. Afterwards, the amount of features is increased again by another 1×1 convolution before subsequently adding the tensor to the shortcut. Similar is done in SqueezeNet [63] which uses 1×1 convolutions to reduce the

number of feature maps before the tensors are subsequently forwarded to a parallel 1×1 and 3×3 convolution, respectively. The SqueezeNet architecture also avoids fully-connected layers by directly using the output of global average pooling as input to the softmax function. In addition, they compress the model using deep compression [51]. The successor of the InceptionNet architecture [64] further reduced resource requirements by proposing spatially separable convolutions where a $K \times K$ convolution is split into a $K \times 1$ convolution followed by a $1 \times K$ convolution.

The most influential architecture is MobileNet [65], which heavily relies on depthwise-separable convolutions where a standard convolution in split into a depthwise convolution and subsequent 1×1 convolution. Depthwise convolutions use a $K \times K$ filter for each feature map separately without considering the cross-channel correlations and the 1×1 convolution then aggregates information across channels. Depthwise-separable convolutions are less expressive than standard convolutions, however, they are more computation and parameter efficient. Certain drawbacks of this convolution type are discovered and discussed in Chapter 7 of this work. Sandler, Howard, Zhu, *et al.* [66] combined the idea of depthwise separable convolution and bottleneck structures and proposed the inverted residual structure. Here, 1×1 convolutions are used to increase the number of feature maps before the computational cheap $K \times K$ depthwise convolution is applied. The number of feature map is subsequently reduced by a 1×1 convolution before the shortcut is added. This inverted structure is more computation and parameter efficient and, consequently, is the most used building block for resource-efficient architectures.

Depthwise convolution can also be seen as group convolutions with group size of one. Group convolutions were initially introduced in the AlexNet [10] architecture to enable model parallelism on two GPUs. The basic idea of group convolutions is to split the feature tensor into several groups and perform independent convolutions on each group, before the outputs of these groups are then stack again to a single tensor. Group convolution significantly reduce the computation and parameter requirements, however, do not consider cross-group correlations. In order to resolve this, Xie, Girshick, Dollár, *et al.* [67] proposed to use group convolutions within residual bottleneck layers, where cross-feature correlations are detected in the subsequent 1×1 convolution. They use a constant split of feature maps throughout the model, however, Radosavovic, Kosaraju,

Girshick, *et al.* [68] found that a constant group size is more beneficial.

While most works use 1×1 convolutions in order to detect cross-channel correlations after group (or depthwise) convolutions, Zhang, Zhou, Lin, *et al.* [69] propose to use shuffle operations instead. The idea is to employ channel shuffle operations after group convolutions to recover the interaction between different groups.

All of the techniques described in this sections share the goal of reducing memory and computation demands of a model. However, they do not reduce activation demands, which can be crucial for accelerating inference and training. This work focuses on all of these metrics in order to generate architectures, which are not only parameter and computation efficient but also fast on hardware.

Neural Architecture Search

Neural architecture search (NAS) is an emerging field within automated ML with a focus of automatic hyperparameter optimizations to maximize prediction performance of models. In more detail, NAS algorithms aim to find optimal solutions for architecture design hyperparameters, such as kernel size, number of features or resolution configurations. This is done by minimizing the validation error over architecture modifications, resulting in an extremely time consuming process since many training runs are required. In addition, the design space of DNNs is usually of exponential size in the number of layers. In order to reduce the required design space, NAS algorithms typically leverage building blocks and design principles developed by manual architecture design and have been proven successful. For instance, most recently proposed NAS algorithms use the depthwise-separable convolutions together with inverted residual block, due to their parameter and compute efficiency.

NAS was initially proposed by Zoph and Le [70] using a technique to encode neural architectures of arbitrary depth as sequences of tokens, sampled from a controller RNN. The controller is trained using reinforcement learning with validation error as a reward signal, in order to generate architecture that achieve good generalization performance. While this technique is able to achieve excellent prediction performance, the training effort is enormous due to the many training runs. The required training time is the limiting factor for larger datasets, which was partly solved by subsequent NAS techniques, e.g., in [71].

A variation of the initial NAS algorithm was proposed by Tan, Chen, Pang, *et al.* [72], which additionally considers the latency of the sampled models on mobile devices. This step increases the required training time even more, however, they train the sampled models for five epochs and select only the best performing models for more epochs.

ProxylessNAS [73] proposed a different technique that aims to reduce the training effort by avoiding a controller: they use a heavily over-parameterized model with parallel path of different architecture blocks. Motivated by binary quantization, they train the selection of a path by backpropagation using the STE and keep only the selected path for each layer. In addition, they also consider latency on various devices, such as (mobile) CPUs and GPUs, and implement a differentiable regularizer in the cost function. This technique is promising because of the low training effort as well as practical inference benefits, however, the limiting factor is the available memory that needs to fit the over-parameterized model.

In order to resolve this memory bottleneck, Stamoulis, Ding, Wang, *et al.* [74] proposed to use a single-path instead of multi-path NAS. All operations are combined in a single superblock where each operations uses a subset of the superblock. The operation selection is done using trainable parameters that determine thresholds for magnitude-based operation selection. Again, the STE is used to approximate the threshold function during training.

Tan and Le [75] proposed a compound scaling technique for discovering parameter and compute efficient architectures. First they identify a small model with high prediction quality using NAS, which is muss less compute intensive than identifying large models. Then, they simultaneously increase number of layers and feature maps as well as spatial resolution, in order to find compound scaling solutions.

The drawbacks of the discovered architectures using the compound scaling technique was analyzed by Radosavovic, Kosaraju, Girshick, *et al.* [68]: the authors found that the resulting architectures are parameter and compute efficient, however, require an extremely high amount of activations, which results in low training and inference performance. They proposed to design network design spaces by parameterizing populations of architectures, which is analogous to manual design of architectures, but elevated to the design space level. The best performing models using their evaluation show interesting design principles that

are in contrast to other models found with NAS.

Liu, Sun, Zhou, *et al.* [76] analyzed the implications of pruning by replicating various techniques and training setups. They found that the common process of training, pruning and fine-tuning is often not necessary for better performance and only the discovered sparsity pattern is important. Consequently, training models with the discovered sparsity patterns from scratch results in similar performance as fine-tuning after pruning. Thus, pruning can also be seen as a technique for NAS.

Chapter 4

Quantized Inference

Quantization is among the most promising compression techniques for reducing memory requirements as well as accelerating inference. There exists a large variety of quantization and implementation techniques, ranging from rather conservative representations (such as half-precision floating point or 8-bit integer) to extreme forms of quantization (such as binary or ternary representations).

This chapter gives detailed insights into quantized inference on various hardware platforms, in order to gain an in-depth understanding of the potential as well as implications of low-precision representations. First, low-precision arithmetic is studied in Section 4.1 on the example of CPUs and an extension of the Eigen library is presented, enabling various representations on ARM architectures. Section 4.2 evaluates QNN inference on ARM CPUs using the library extension together with popular and accurate quantization techniques. Then, QNNs are trained, mapped and evaluated on FPGAs using the FINN data-flow architecture in Section 4.3. Section 4.4 studies the efficiency of relatively conservative quantized representation on general-purpose GPUs. Last, a real-world application is presented in Section 4.5, which highlights the benefits and implications of quantization. This chapter extends and summarizes three publications[1]–[3].

4.1 Low-Precision Signed-Integer GEMM

Latency and energy efficiency of DNNs mainly depend on the optimizations of matrix-vector and matrix-matrix multiplication subroutines. The increasing demand for these algorithms in scientific computing motivated explorations of highly effective algorithmic optimizations, such as parallelization, vectorization,

caching and many more. These algorithms are usually wrapped into Basic Linear Algebra Subroutines (BLAS) libraries, which are developed specifically for a certain processor and can be used without any knowledge of the underlying hardware or the expertise of algorithmic optimizations.

The prime example for the usage of BLAS libraries are ML frameworks (i.e. Theano or TensorFlow), where users are able to develop hardware-agnostic code in Python and, during execution, the code generation is performed using the BLAS library Eigen for Intel and ARM CPUs or cuBLAS for NVidia GPUs. While this software flow enables developers an extremely productive working environment with highest performance, it also restricts research to the available features of the respective framework. In the context of resource-constraint DNN inference, the main missing feature are low-precision formats, since BLAS libraries are usually developed for single-precision or double-precision floating-point format.

This section introduces an extension for the Theano framework using the Eigen library, that offers signed-integer formats from 32-bit to binary precision. Enabling such formats allows easy deployment and explorations of quantized neural networks. The methodology is exemplified and evaluated using the ARM Instruction Set Architecture (ISA), however, can be extended to other hardware as well. This work has been published at the *Workshop on UnConventional High Performance Computing 2017 (UCHPC 2017), in conjunction with EuroPAR 2017* [1].

4.1.1 Methodology

There are two possible ways to enable low-precision matrix multiplication in ML frameworks: the first one is to implement code from scratch and accelerate it for a certain processor using compiler and algorithmic optimizations. The second way is to use already optimized code and extend it for low-precision formats, which is done in this work on the example of the Eigen library. The methodology introduced in this section allows leaving most of the BLAS algorithm untouched and only adapting the lowest level of the subroutine. This is not only beneficial in terms of productivity but, more importantly, extremely important for the performance since BLAS algorithms contain thousands lines of complicated code, which are developed over a long period of time and achieve up to theoretical peak performance of the respective processor.

BLAS libraries feature matrix-matrix and matrix-vector multiplication algorithms, where elements of the resulting matrix $\mathbf{C} \in \mathbb{R}^{I \times J}$ of two input matrices $\mathbf{A} \in \mathbb{R}^{I \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times J}$ are calculated as

$$c_{i,j} = \sum\nolimits_{n=1}^{N} a_{i,n} \cdot b_{n,j}. \tag{4.1}$$

Consequently, the BLAS subroutines iterates over the input matrices and applies a large amount of Multiply-Accumulate (MAC) operations. The performance of these MAC operations highly depend on ths use of Single Instruction Multiple Data (SIMD), which is a vectorization technique that enables the computation of multiple data elements with a single instruction. SIMD exploits the low cost of data-level parallelism in CMOS processes and are ubiquitous in current architectures from powerful server CPUs to tiny microcontrollers.

The vectorization for matrix multiplications can be realized by either computing different scalar products in parallel using broadcasting of single elements along the SIMD lane or by computing single scalar products in parallel and reducing the final results. The majority of BLAS implementations, including the Eigen library, applies the broadcasting technique: let $S = \frac{S_{length}}{O_{width}}$ be the number of operands with a bit width of O_{width}, within a SIMD lane of bit length S_{length}, then the elements of $\mathbf{C}$ can be calculated in parallel as

$$c_{i,j}, ..., c_{i,j+S} = \underbrace{\sum\nolimits_{n=1}^{N} a_{i,n} \cdot b_{n,j}, ..., \sum\nolimits_{n=1}^{N} a_{i,n} \cdot b_{n,j+S}}_{\text{1st SIMD stage}}, \tag{4.2}$$

where $a_{i,n}$ is broadcasted along the SIMD lane. Vectorization could be also implemented by simply computing elements of $\mathbf{A}$ and $\mathbf{B}$ in parallel, however, the broadcasting does not require a reduction within the SIMD lane, which is much faster in practice.

While the broadcasting works efficient for floating-point arithmetic where the bit width of intermediate registers can be kept constant through normalization, it can not be applied to reduced-precision arithmetic in its basic form. Low-precision formats are usually represented as integers (i.e. $int8$) that rely on careful adjustments of intermediate and resulting registers. When multiplying or adding two $int8$ values, the intermediate register has to be doubled to $int16$ or increased by one bit to $int9$, respectively, in order to guarantee no overflows during the calculation. Consequently, the SIMD scheme of Equation 4.2 needs

to be adopted accordingly: let $A = \frac{S_{length}}{A_{width}}$ be the number of accumulators with a bit width of A_{width} and $W = \frac{A_{width}}{O_{width}}$ the number of reduced-precision operands with a bit width of O_{width} within a SIMD lane of bit length S_{length}, then the scalar products $c_{i,j}$ can be calculated as

$$c_{i,j}, \ldots, c_{i,j+A} = \underbrace{\sum_{n=1}^{N/W} \overbrace{\sum_{w=1}^{W} a_{i,n+w} \cdot b_{n+w,j}}^{\text{2nd SIMD stage}}, \sum_{n=1}^{N/W} \overbrace{\sum_{w=1}^{W} a_{i,n+w} \cdot b_{n+w,j+A}}^{\text{2nd SIMD stage}}}_{\text{1st SIMD stage}}. \tag{4.3}$$

This step adds the *2nd SIMD stage* where low-precision operands are multiplied into product registers of length $2 \cdot O_{width}$ and afterwards W times reduced into register of length A_{width}. The whole process is illustrated in Figure 4.1 on the example of *int*8 operands: multiplications in combination with reduction refer to the *2st SIMD stage* and accumulations refer to the *1st SIMD stage*.

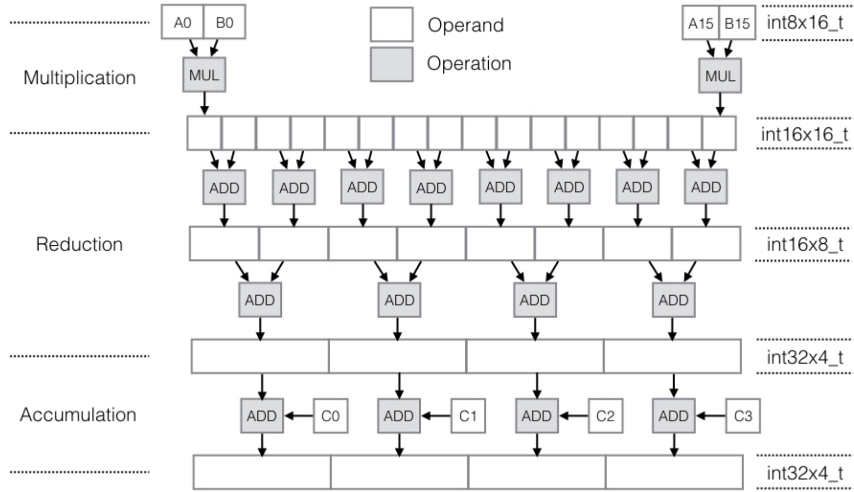


Fig. 4.1 Simplified illustration of the MAC operation for int8_t representation. [1]

Introducing these two SIMD stages is beneficial for multiple reasons: first, the BLAS algorithm can be seen as a black box without consideration about high-level optimizations but all performance benefits of them. Second, the SIMD lane is leveraged optimally since both, broadcasting (*1st SIMD stage*) and reduction (*2nd SIMD stage*) are combined with their respective strengths. And last, only the MAC operation is required to be adopted to the respective operand's bit width.

4.1.2 MAC Implementations

The methodology described in Section 4.1.1 is processor-agnostic and applicable to a wide range of computing systems. However, the MAC implementation highly depends on architectural features as well as available instructions and data types. This section describes several MAC implementations for low-precision formats using the ARM architecture, that dominates many domains of embedded computing today and is therefore a prime candidate for DNN inference on edge devices. The used data types are the low-precision signed integers $int16$, $int8$, $int4$, $int2$, and $int1$ as well as the full-precision types $int32/float32$ for comparison. ARM's NEON vector extension is used for SIMD realization which allows 128-bit vectorized instructions.

The implementation of single-precision floating point MAC is straight forward: three vectors (**a**, **b** and **c**), each containing four float scalars, are simply forwarded to ARM's MAC instructions and the result can be returned.

```
1  float32x4_t macf32( float32x4_t a, float32x4_t b, float32x4_t c )
2  {
3    // Multiply and Accumulate: a*b+c
4    return vmla_f32( a, b, c );
5  }
```

This MAC routine can be consequently implemented using a single instruction without the need of data conversion or bit-width adjustment. To enable other formats, more sophisticated instructions are required: Table 4.1 summarizes the used instructions for reduced-precision computations, such as multiplication with bit-width doubling, accumulation and vector reduction as well as bit-wise operations.

int16 and int8 MAC

Although NEON features single MAC instructions for $int8$ and $int16$ formats, they can not be applied within the low-precision MAC routines, because they do not allow doubling the bit width during multiplication. The *VMULL* instruction is used instead that doubles the operand's bit width during the multiplication to $int16$ and $int32$, respectively. The multiplication of the highest and lowest 64-bit vectors of input vectors **a** and **b** are performed sequentially, since instructions with bit-width doubling are constraint to 64 bit operands. For the $int16$ MAC

Table 4.1 Instruction overview for the MAC operation.[1]

Operation	Instruction	Description
Multiplication	VMLA	Multiplies the elements of two vectors and accumulates the elements of a third vector - Supports 32/16/8 bit.
	VMUL	Multiplies the elements of two vectors - Supports 32/16/8 bit.
	VMULL	Multiplies the elements of two vectors and doubles the bit width - Supports 32/16/8 bit.
	VAND + VEOR	Bitwise logic instruction - Supports 32/16/8 bit.
Reduction	VPADDL	Adds adjacent pairs of elements of a vector - Supports 32/16/8 bit.
	VPADAL	Adds adjacent pairs of elements of a vector and accumulates the result by elements of a second vector - Supports 32/16/8 bit.
	VCNT	Counts the number of set bits of a vector - Supports 8 bit.
Accumulation	VADD	Adds the elements of two vectors - Supports 32/16/8 bit.

implementation, the resulting vectors are summed up into a single vector before the result is accumulated to vector **c**

```
1  int32x4_t macs16( int16x8_t a, int16x8_t b, int32x4_t c ) {
2    // Vector extraction
3    int16x4_t high_a =  vget_high_s16( a );
4    int16x4_t high_b =  vget_high_s16( b );
5    int16x4_t low_a = vget_low_s16( a );
6    int16x4_t low_b = vget_low_s16( b );
7    // Multiply
8    int32x4_t high_r = vmull_s16( high_a, high_b );
9    int32x4_t low_r = vmull_s16( low_a, low_b );
10   // Reduce
11   int32x4 r = vadd_s32( high_r, low_r );
12   // Accumulate
13   return vadd_s32( r, c );
14 }
```

As can be seen, the $int16$ MAC routine requires eight instructions because of bit-width adjustments, while the $float32$ MAC can be computed within a single instruction.

The *int*8 MAC implementation is very similar to *int*16, but requires an additional reduction stage where adjacent pairs of elements are added before accumulation.

```
1  int32x4_t macs8( int8x16_t a, int8x16_t b, int32x4_t ) {
2    // Vector extraction
3    int8x8_t high_a =  vget_high_s8( a );
4    int8x8_t high_b =  vget_high_s8( b );
5    int8x8_t low_a = vget_low_s8( a );
6    int8x8_t low_b = vget_low_s8( b );
7    // Multiply
8    int16x8_t high_m = vmull_s8( high_a, high_b );
9    int16x8_t low_m = vmull_s8( low_a, low_b );
10   // Reduce
11   int32x4_t high_r = vpadll_s16( high_m );
12   int32x4_t low_r = vpadll_s16( low_m );
13   int32x4 r = vadd_s32( high_r, low_r );
14   // Accumulate
15   return vadd_s32( r, c );
16 }
```

Thus, the *int*8 MAC routine requires two more instructions for the additional reduction stage than the *int*16 implementations. The whole MAC sequence is illustrated in Figure 4.1 on the example of *int*8 operands. This results into an instruction increase of 2x and 1.25x for *int*16 and *int*8, respectively, in comparison to $float32$ MAC, which is in contrast to the assumption that low-precision formats can be used to accelerate computations. However, the low-precision implementations have two favourable properties: first, their instructions use simpler logic and can be computed in fewer cycles. Second, the reduced operand's bit width requires fewer memory accesses and, consequently, fewer cycles for loading the data.

int4 MAC

While *int*8 and *int*16 formats are supported inherently by NEON, it lacks support for *int*4 formats. In particular, the extraction of *int*4 values from the input vectors causes a high instruction overhead. In order to perform the extraction, the even and odd indexed *int*4 values are masked out from the 128-bit input vectors **a** and **b** via bit-wise logic operations and the values are split into two separate 128-bit vectors. Once the extraction is done, the obtained vectors can

be simply multiplied without bit-width doubling. Last, a three-layer reduction is performed before the resulting vector elements are summed up.

int2 MAC

The multiplication of the $int2$ MAC is realized by evaluating the resulting positive and negative values separately via bit-wise logic operations (AND, XOR). Then, a 8-bit population count is performed to count the positive and negative values within a 8-bit vector. The resulting positive values are subtracted by the resulting negative values. Two reduction levels transform the $int8$ representation into a $int32$ intermediate representation and accumulate the vector by elements of input vector **c**.

int1 MAC

For the $int1$ MAC, the proposed technique by Courbariaux et al. [30] is used: the basic idea is to replace the actual multiplications of input vectors **a** and **b** with bit-wise XNOR operations and perform the reduction via population count as:

$$\mathbf{a} \cdot \mathbf{b} = N - 2 \cdot popc(xnor(\mathbf{a}, \mathbf{b})), \tag{4.4}$$

where $xnor$ is the bit-wise logic operation, $popc$ denotes the counting of set bits and N is the length of vectors **a** and **b**. Since NEON includes only a 8-bit population count, two further reduction levels are used in order to reduce the results into a $int32$ intermediate representation. The result is subsequently accumulated by input vector **c**.

```
1  int32x4_t macs8( int8x16_t a, int8x16_t b, int32x4_t ) {
2    // Multiply
3    int8x16_t m = veor_s8( a, b);
4    // Reduce
5    int8x16_t r0 = vcnt_s8( m );
6    int16x8_t r1 = vpadll_s( r0 );
7    // Accumulate
8    return vpadal_s16( r1, c );
9  }
```

This routine requires ultimately four instructions, reducing the overall instruction by 8x in comparison to $float32$ MAC. The $int1$ MAC is illustrated in Figure 4.2 (left).

Bit-Serial MAC

The binary MAC technique can be generalized to any bit combination through the bit-serial MAC by iteratively computing binary MACs for each bit combination as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{n=1}^{N} \sum_{m=1}^{M} 2^{n+m} \cdot popc(and(\mathbf{a}_n, \mathbf{b}_m)), \tag{4.5}$$

where N and M are bit widths of operand **a** and **b**, respectively. While this MAC technique greatly suits the demands of adaptive bit widths, it comes at the cost of increased latency through instruction serialization.

4.1.3 Optimizing reduction depth

As discussed in Section 4.1.1, low-precision MAC operations consist of multiplication, reduction and accumulation, where the respective reductions require the most time. The number of reductions within the MAC depends on the bit width of operands as well as accumulators. Hence, the only way of reducing the reduction time is to reduce the accumulator bit width, however, almost all low-precision formats can not avoid 32-bit accumulators in order to avoid overflows.

Binary and bit-serial computations differ because the multiplication produces again binary results that are either -1 or +1. As illustrated in Figure 4.2 (left), the first reduction layer of binary MAC is performed via 8-bit population count, followed by two integer reductions before the result is added to the accumulator. Considering that the scalar product of a row and a column vector takes N (matrix depth) accumulations of a maximum value of 8, the maximum scalar value is $N \cdot 8$ for the first reduction layer and $N \cdot 16$ for the second reduction layer. Therefore, a reduced bit width ($Width$) for the intermediate representation is sufficient if $N < \frac{2^{Width}}{Width}$ holds.

Consequently, the GEMM implementation can be modified for binary input representation to dynamically adapt among 32-bit, 16-bit, and 8-bit accumulators by only evaluating the matrix' depth. As a result, 16-bit accumulators require one reduction layer less (see Figure 4.2 middle) and 8-bit accumulators require two reduction layer less (see Figure 4.2 right) , compared to 32-bit accumulators. Obviously, the resulting representation of this optimization differs from the expected output representation. Thus, the last MAC operation of the scalar

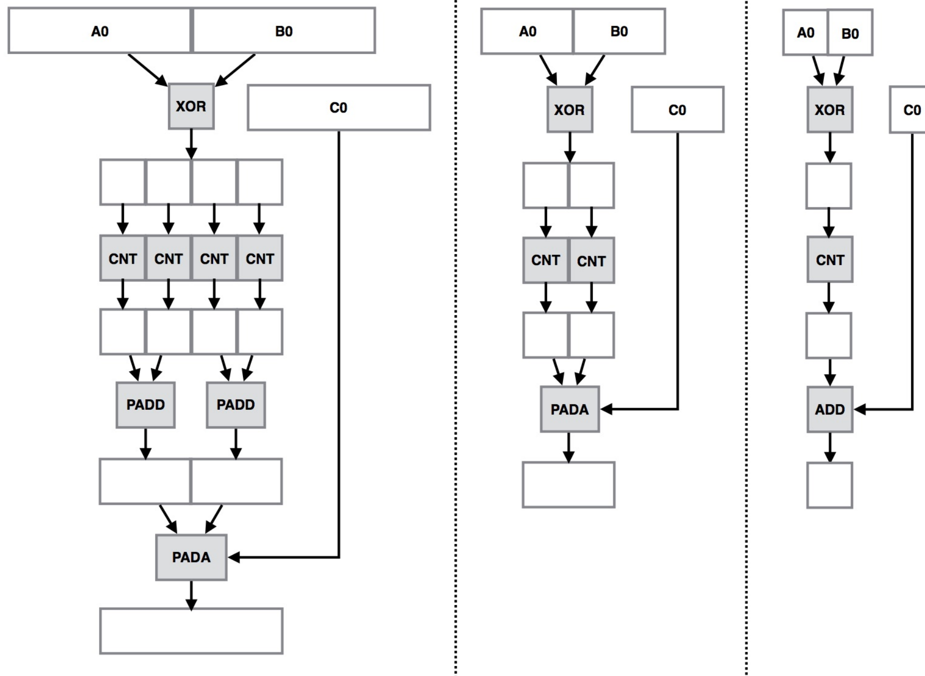


Fig. 4.2 Illustration of the binary MAC operation for 32-bit (left), 16-bit (middle) and 8-bit (right) accumulators.

product of a row vector and a column vector has to reduce the intermediate representation to a 32-bit output representation. As a result, the computational complexity of the reductions can be reduced from $O(n^2)$ to $O(n)$ which directly translates into a significant performance improvement for matrices with small ($N < 32$) and mid-sized ($N < 4096$) depths.

4.1.4 Performance Analysis

This section evaluates execution time and memory footprint of the low-precision signed-integer extension for the Eigen library. The obtained results are compared to the single-precision matrix-multiply routine of the original Eigen library on a system with a 2.32 GHz ARM quad-core Cortex-A15 CPU. Table 4.2 reports the execution time of various bit widths and matrix sizes.

The expected execution time of the core code sequence can be estimated using instruction latency data from the ARM Technical Reference Manual [77]. Table 4.3 summarizes the estimated and measured speed-up of the GEMM operator for different input representations in comparison to single precision. As can be seen, the measured speed up roughly matches the estimations with some minor deviations: i) Most observed speed-ups are actually higher than estimated, due

Table 4.2 Summary of the obtained results: execution time and speed-up (SU) over $float32$ format.

Size	Metric	$float32$	$int16$	$int8$	$int4$	$int2$	$int1$
128	Time	0.18ms	0.37ms	0.16ms	0.22ms	0.12ms	0.05ms
	SU	1.00	0.48	1.06	0.81	1.43	3.61
256	Time	1.34ms	2.94ms	1.26ms	1.66ms	0.44ms	0.08ms
	SU	1.00	0.46	1.07	0.85	3.10	17.09
512	Time	11.54ms	24.02ms	10.03ms	12.92ms	3.27ms	0.54ms
	SU	1.00	0.48	1.15	0.89	3.52	21.17
1024	Time	90.10ms	192.08ms	81.63ms	104.03ms	26.17ms	4.73ms
	SU	1.00	0.47	1.11	0.87	3.43	18.99
2048	Time	0.70s	1.52s	0.61s	0.83s	0.20s	0.04s
	SU	1.00	0.46	1.10	0.85	3.45	19.81
4096	Time	5.53s	12.15s	5.10s	6.57s	1.60s	0.26s
	SU	1.00	0.46	1.09	0.84	3.43	20.83
8192	Time	44.72s	97.36s	40.88s	52.53s	12.74s	3.11s
	SU	1.00	0.45	1.10	0.85	3.50	14.34

Table 4.3 Expected and actual speed up of the signed-integer GEMM derived from the required cycles of the MAC operation.

Input Rep.	Cycles	Estimated Speed-Up	Observed Speed-Up
$int32$	6	1	1
$int16$	30	0.40	0.45-0.48
$int8$	36	0.67	1.06-1.15
$int4$	69	0.70	0.81-0.89
$int2$	39	2.46	1.43-3.52
$int1$	15	12.80	3.61-21.17

to additional memory savings of the operator's inputs. ii) Speed-ups of small matrices ($< 256 \times 256$) combined with 1-bit and 2-bit formats are lower than estimated, since Eigen enforces padding of small matrices.

Figure 4.3 illustrates the average speed-up of the low-precision implementations (using integer logic and bit-serial technique) and memory reduction. Both techniques achieve similar latency between 1-4 bit, however, bit serial performs slightly better for 2-bit and 4-bit computations, but it performs clearly worse than integer logic for configurations that exceed 4 bit. In general, there is no latency improvements for ≥ 4 bit with the exception of 8-bit formats, which slightly outperforms floating-point computations. Besides the additional reduction overhead, this is mainly due to instruction serialization caused by bit-width doubling when the multiplication is performed. On the other hand, implementations using ≤ 3

bit for computations achieve relatively low latency, which is inline with extreme forms of quantization.

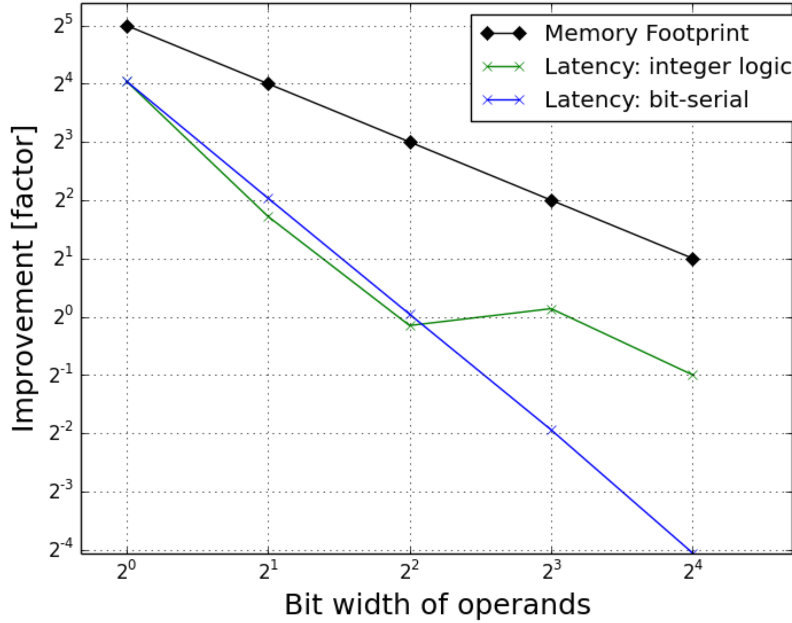


Fig. 4.3 Memory footprint and execution time of 32-bit and reduced-precision signed integer GEMM.

4.2 QNNs on CPUs

As shown in the previous section, quantized operations can significantly accelerate inference on general-purpose CPUs using either integer or bit-serial arithmetic. While these results are promising, it is a necessity to additionally evaluate the prediction performance of neural networks within the targeted quantization regime. The following sections explore the impact and implications of several quantization techniques on a variety of tasks and datasets.

4.2.1 MLP on MNIST

This section evaluates the efficiency of binarization on ARM processors using an MLP consisting of 3 hidden layers of 4096 units on the MNIST task. The experimental setup and binarization technique follows the original implementation of Hubara et al. [30] using the Theano framework. Binarization is applied to all fully-connected layers with exception of the input layer, as is sensitive to such

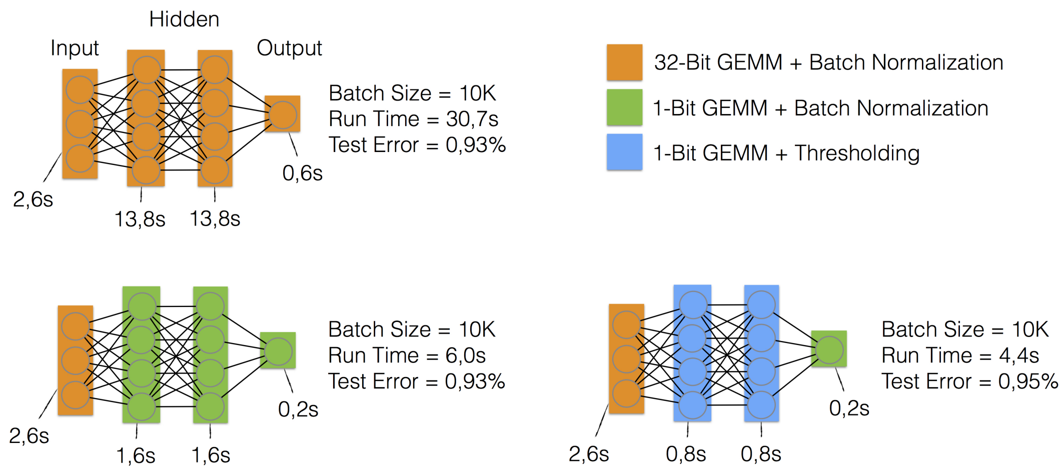


Fig. 4.4 Runtime comparison of real-valued and binarized MLPs on the MNIST task.

extreme forms of quantization. The low-precision Eigen extension is integrated into Theano as a customized operator, in order to benchmark inference latency of binarized neural networks.

Figure 4.4 reports error and runtime for the 10 thousand test samples of the respective models. As can be seen, the two hidden layers of the real-valued model require by far the most runtime, whereas input and output layers are relatively fast. Through binarization it is possible to reduce the runtime of hidden layers by a factor of 8.6×, which improves the overall runtime by a factor of 5.1× without reducing the prediction performance.

The fully-connected layers use batch normalization and, subsequently, apply the sign function as non-linearity. This normalization requires additional memory accesses as well as floating-point computations, resulting in a high impact on execution time in comparison to the light-weight binarized computations. Umuroglu et al. [22] showed that, for binarized neural networks, the same output can be pre-computed using the parameters of the batch normalization layer, which can be used as threshold parameter to determine the output activation. Applying this thresholding technique in the evaluated binarized MLP further reduces the runtime of hidden layers by a factor of 17.3× and the overall runtime by a factor of 7.0×.

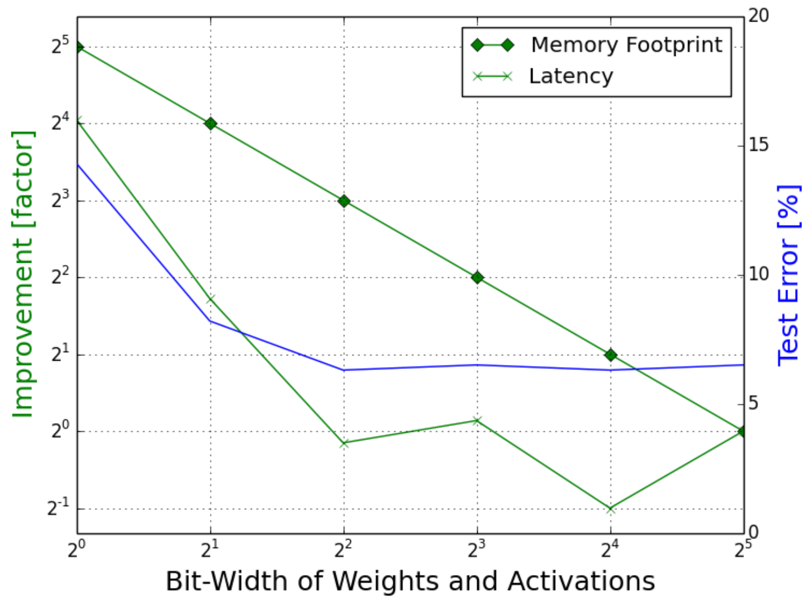


Fig. 4.5 Improvement of reduced precision over single-precision floating-point on memory footprint and latency (green) and the respective test error of ResNet-32 on CIFAR-10 (blue).

4.2.2 Prediction Accuracy, Memory Footprint and Latency

Results obtained in the previous section indicate efficient compression and acceleration potential through binarization without resulting in accuracy degradation. However, the MNIST task is relatively simple to solve and the excellent results might not be representative for other more complex task. This section evaluates low-precision inference on ARM CPUs using the example of the CIFAR-10 task, which is considerably harder to solve than MNIST. A ResNet variant is used as neural architecture with 32 layers and, again, the first layer is not quantized. The quantization framework of Zhou et al. [41] is used for training the low-bit models, where weights and activations use equally defined bit widths. The latency improvement, in comparison to the single-precision floating point implementation, is estimated using the average improvements of Section 4.1.4.

Figure 4.5 reports improvement factors of memory and latency (green) in comparison to the floating-point model as well as test error (blue) of the various models. Whereas binarization works without considerable accuracy loss on the MNIST task, it results in a severe degradation on the CIFAR-10 tasks.

This ultimately indicates the performance implications when binarization is applied to real-world applications, since such extreme accuracy losses are not acceptable. The model gains a substantial amount of accuracy when the bit width of activations and weights is increased to 2 bit. The inference latency of 2-bit models is about 4× slower than binarized models, however, there is still a speedup of 2× in comparison to the floating-point model. While 2-bit models are a good trade-off between inference performance and accuracy, only 4-bit models achieve single-precision accuracy. However, 4-bit (or higher bit) models do not achieve faster inference potential on these kind of processors.

4.2.3 Analyzing the Impact of Quantization Techniques

The key insight from previous section is that models with < 4 bits achieve highest inference performance while models with ≥ 4 bits result in real-valued accuracy. Further acceleration of ≥ 4 bit computations is difficult, because of the instruction and processor limitations. On the other hand, there is improvement potential in regard of optimizing quantization techniques for < 4 bit representations.

This section studies the impact of several popular quantization technique on model accuracy in the hardware-efficient range of $[1, 3]$ bit. The techniques explored here are DoReFa-Net [41], BNN [30], BWN [29], TTQ [32] and LQ-Net [40]. TTQ is modified to uniform quantization in order to be inline with the low-bit inference library, while the original TTQ implementation leverages non-uniform weight representations. For this experiment, the DNNs are quantized in the three modes (i) weight only, (ii) activation only, and (iii) combined weight and activation quantization, respectively. However, some quantization approaches are designed for a particular mode, e.g., BWN and TTQ only consider weight quantization whereas BNN only considers combined weight and activation quantization. For combined quantization the same bit widths is used for the weights and the activations. The data complexity is increased from CIFAR with 10 to 100 classes, in order to better highlight the performance differences of the various techniques. Furthermore, a more parameter and computation efficient DenseNet [15] variant is used with bottleneck as well as compression layers and a depth of a 100.

Figure 4.6 reports test accuracy for different bit setting and indicates several interesting quantization aspect: i) As expected, the test error decreases gradually

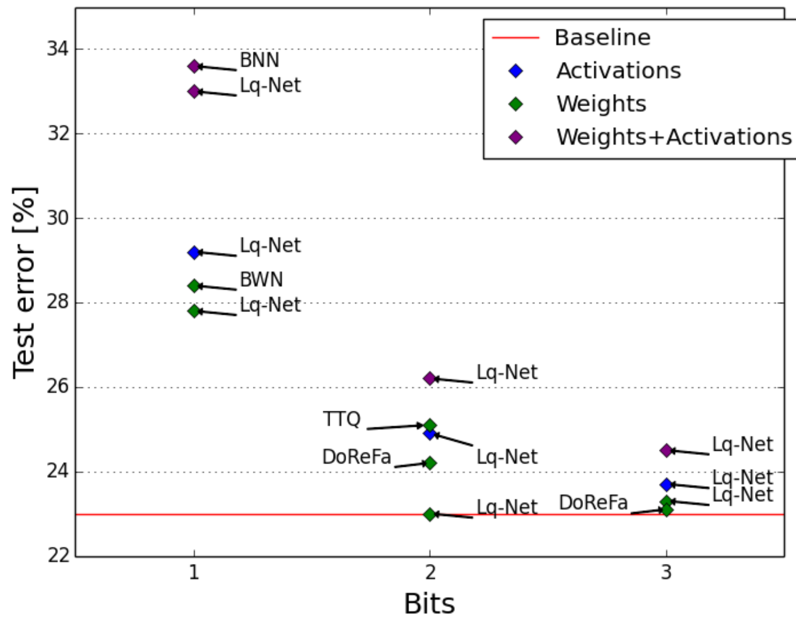


Fig. 4.6 Comparison of several popular quantization techniques using the DenseNet-BC-100 architecture trained on the CIFAR-100 dataset. The horizontal red line shows the error of the real-valued baseline. Quantization is performed using different bit widths in the three modes activation only (blue), weight only (green) and combined weight and activation quantization (purple), respectively.

with increasing bit widths for all quantization modes and for all quantization approaches. ii) Prediction performance is more sensitive to activation than to weight quantization. iii) Weight and activation quantization interfere each other, which means that combined quantization achieves less accuracy than the worst single-mode quantization. The best overall performing quantization is LQ-net [40], which outperforms other techniques throughout all bit combinations. However, this accuracy efficiency comes at the cost of much longer training time: the time per iteration increases by a factor of up to 4.6 (depending on the bit width) for LQ-Net, ultimately resulting in a trade-off between training and inference time.

4.3 QNNs on FPGA

The prime platform for QNNs are data-flow architectures on FPGAs, where the main objective is to keep all required data for inference in on-chip memory.

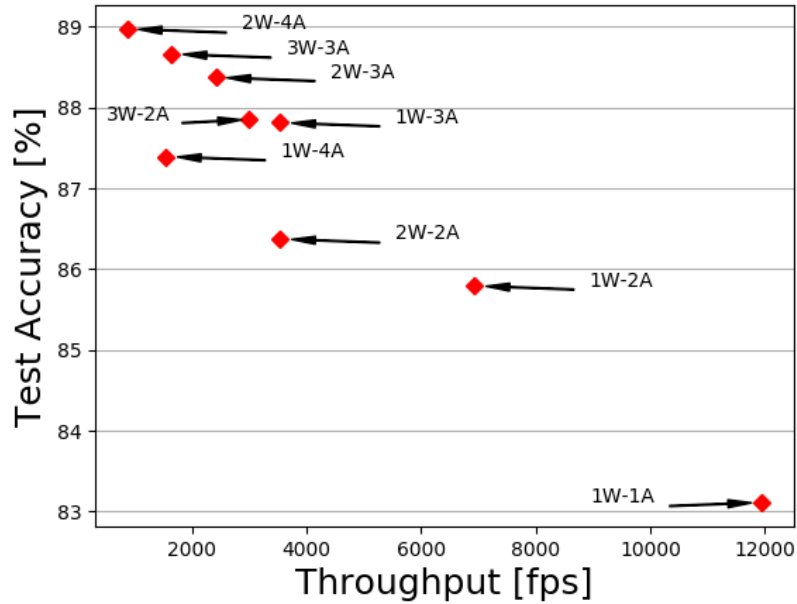
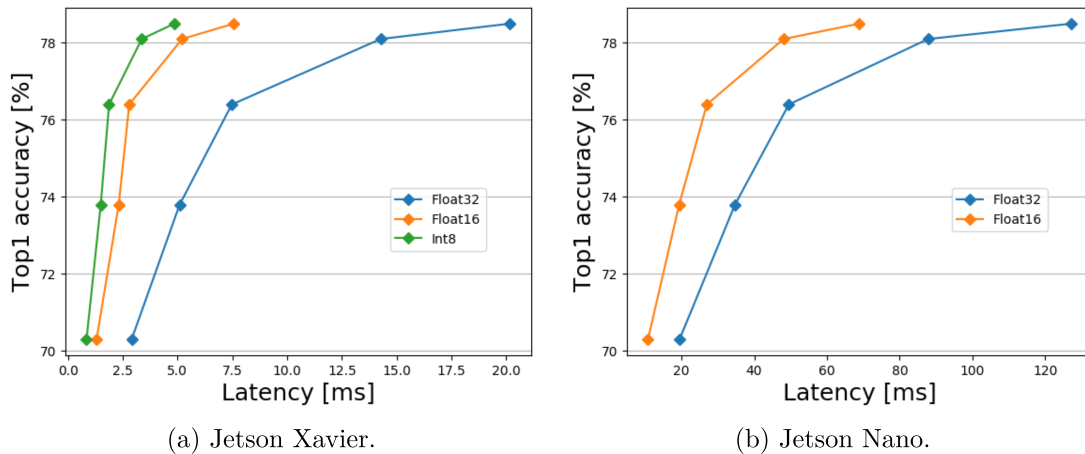


Fig. 4.7 Throughput and accuracy on a Xilinx Ultra96 board using varying bit width on the CIFAR-10 task.

Staying entirely on chips enables inference in a pipelined fashion, resulting in high-throughput and low-energy performance. Furthermore, the low-resource requirements of bit-serial compute units allows for a high amount of parallelism.

This section evaluates QNNs on FPGAs using the FINN framework [22] for generating data-flow architectures on reconfigurable hardware. Figure 4.7 shows test accuracy over throughput for the FINN framework using several bit combinations on a Xilinx Ultra96 board. The CIFAR-10 tasks is used for evaluation on a variant of the VGG architecture and the reported results were produced within the Master's thesis of Hendrik Borras. In order to guarantee a reasonable comparison of the varying bit combinations, the configuration of the FINN framework is adjusted so that highest throughput is targeted with respect to the available resources (BRAM, LUTs, etc.) of the device.

As expected, the test accuracy increases gradually with additional bits while the throughput decreases accordingly. The Pareto front indicates, that the best performing models use a combination of 1 bit for weights and a gradual increase of activations to ≤ 3 bits. Afterwards the models perform best if weights are scaled to 2 bits and activation bit width is further increased to 4 bits. This seconds the observation of the previous section, which shows that model performance is sensitive to activation rather than weight quantization.



(a) Jetson Xavier.

(b) Jetson Nano.

Fig. 4.8 Latency and accuracy on the ImageNet task using different supported formats on NVidia boards.

4.4 QNNs on GPU

FPGAs feature reconfigurable hardware and offer excellent opportunities for bit-serial computations within a data-flow architecture. This technique, however, requires all data to be on chip, possible preventing larger models to be deployed or the use of higher bit-width formats. On the contrary, on-chip memory is a scarce resource on GPUs but the latency-tolerating programming model enables high compute and data throughput. As a consequence, GPUs are capable of deploying very large models while also achieving high utilization of the massively-parallel cores. The drawback of GPUs is that inference performance depends on the use of BLAS libraries and the built-in data formats, which ultimately constrains the degrees of freedom when deploying quantized models.

State-of-the-art accelerators by NVidia started to support quantized operations in form of rather conservative representation such as half-precision floating point and 8-bit integer. Figure 4.8 shows the latency with respect to accuracy using single-precision, half-precision as well as 8-bit integer formats for weights and activations. The inference latency is measured on NVidia's Jetson Nano and Xavier boards, using the cuDNN library and TensorRT compiler. The reported accuracy refers to single-precision ResNet models (using 18, 36, 50, 101, 152 layers) on the ImageNet task and the same accuracy is reported for half-precision as well as 8-bit integer representations, because no accuracy degradation is assumed.

Half-precision floating point achieves 2.2 - 2.7× faster inference execution on the Xavier board and 1.8 - 1.9× on the Nano board, in comparison to single-precision floating point. Reducing the bit representation to 8-bit integer results into further speedups of 3.4 - 4.2× on the Xavier board. Please note that the Nano board does not feature hardware support for 8-bit integer and, thus, there is no acceleration potential on this platform. These results highlight the importance of supporting quantized operations on general-purpose and domain-specific accelerators as well as their corresponding software stack and libraries.

4.5 Resource Efficient Deep Eigenvector Beamforming

There is an increasing interest in using neural networks in several speech enhancement applications. For instance, they are used in the field of acoustic beamforming for estimating the speech mask. This mask is used to determine the power spectral density (PSD) matrices of the multi-channel speech and noise signals, which are used to obtain Generalized Eigen Vector (GEV) beamformer. While neural networks achieve impressive performances on such tasks, the models are inefficient when deployed in resource-constraint environments due to their computational and memory requirements.

In collaboration with Matthias Zöhrer, the DNN responsible for speech mask estimation is replaced with a BNN and evaluated on an ARM CPU. This work has been published at *International Conference on Acoustics, Speech and Signal Processing (ICASSP)* [3]. The binary GEV beamformer is evaluated using the CHiME4 corpus [78], which provides 2 and 6 channel recordings of a close-talking speaker corrupted by four different types of ambient noise. Table 4.4 reports the Word Error Rate (WER) for the 2 and 6 channel data using DNNs and BNNs with a GEV-PAN beamformer.

Binarization is able to accelerate the 513 neuron models for both, 2 and 6 channel data, by a factor of 11.9×, however, it also results into 7.8% and 1.6% accuracy degradation, respectively. This severe accuracy loss can be compensated by scaling the width of the BNN model from 513 to 1024 neurons, which reduces the gap to the DNN to 3.0% and 1.6%. As a consequence, the resulting BNN models achieve prediction performance close to the real-valued model, but with

Table 4.4 Test WER, runtime and speedup for DNN and BNN models.

	Neurons	WER	Runtime	Speedup
2 channel		%	ms	
DNN	513	27.6	16.7	1.0×
BNN	513	35.4	1.4	11.9×
BNN	1024	30.6	8.1	2.1×
6 channel		%	ms	
DNN	513	18.7	16.7	1.0×
BNN	513	21.2	1.4	11.9×
BNN	1024	20.3	8.1	2.1×

2.1× faster inference.

4.6 Summary

Reducing the resolution of weights and activations through quantization is one of the most promising techniques in order to enable efficient inference on edge systems. However, there is no general guarantee that any type of quantization is advantageous on any type of processor. Unsupported data formats, bad scalability of bit width or accuracy loss may cause performance degradation if the setup is not chosen properly. In this context, the key insights of this section are:

- Binarization greatly reduces memory and inference time, however, can potentially cause severe accuracy degradation of the model.
- There is a general observation that model accuracy benefits more from increasing the bit width of activation rather than weights.
- CPUs can leverage low-bit formats (≤ 3 bit) for faster inference, however, fail at benefiting from higher bit configurations (> 3 bit), due to lack of hardware support.
- FPGAs are excellent at benefiting from quantization by leveraging extreme low-bit representation (≤ 4 bit), bit-serial computations as well as data-flow architectures.

- Modern GPUs and their respective BLAS libraries support conservative representation (i.e. half-precision floating point or 8-bit integer), which shows great acceleration potential while not degrading prediction accuracy.

In summary of this section, it can be stated that quantization shows excellent compression potential for neural networks. The main drawbacks are increased training time, possible accuracy degradation and unsupported data formats. The latter issue, however, is likely to be resolved with upcoming processors technologies that support these formats, which can be seen on the evolving NVidia accelerators.

Chapter 5

Reduce-and-Scale

The previous section explored in detail the potential as well as the implications of quantization on various hardware platforms: extreme low-bit inference is highly beneficial on reconfigurable and general-purpose hardware. The drawback here is that such extreme forms of quantization can lead to severe accuracy degradation. Increasing the bit width without explicit hardware support is not very advantageous on general-purpose processors. However, there is a large availability of these processors already available in our society: for instance, ARM produced about 160 billion chips that are used in edge platforms, such as mobile phones, tablet computers or smart devices.

In order to leverage these widely spread architectures, it is necessary to develop techniques that compress neural networks without sacrificing prediction performance and, at the same time, map well onto such generic systems and platforms. The insights gained in Section 4 indicate that non-uniform outperforms uniform quantization and weights require less bits than activations. Furthermore, weights can contain a substantial amount of sparsity which potentially reduces inference time, since MAC computations with zero operand can be simply skipped. However, the standard matrix-multiplication algorithms - including quantized and bit-serial techniques - can not implement or leverage such representations.

This section introduces the *Reduce-and-Scale (RaS)* technique, a compression and inference algorithm co-design, developed for high accuracy as well as hardware-efficient mapping. RaS exploits non-uniformly quantized weights, including configurable sparsity, and uniformly quantized activations in order to maintain prediction performance. The inference algorithm maps this representation using vectorization and parallelization onto the processor. This work has

been published at the *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD), 2018* [4].

5.1 Quantization

The quantization technique represented in this section is based on 8-bit uniformly quantized activations and non-uniform ternary quantization for weights. In more detail, the weights are trained following the Trained Ternary Quantization (TTQ) [32] technique and activations are mapped to a fixed-point format. This unusual combination aims to maintain the prediction quality while substantially decreasing the memory and computation requirements of the models.

5.1.1 Quantizing Weights

TTQ [32] uses two trainable scaling coefficients W_l^p and W_l^n for each layer l for representing positive or negative weights. Both coefficients are independent, asymmetric parameters and are trained together with their respective weights w. The ternary weights $\widetilde{w}_l^i$ are obtained by applying the thresholding function

$$\widetilde{w}_l^i = \begin{cases} W_l^p : w_l^i > \Delta_l \\ 0 : |w_l^i| \leq \Delta_l \\ -W_l^n : w_l^i < -\Delta_l \end{cases} \tag{5.1}$$

on the real-valued weights w_l^i, where positively and negatively clustered values are assigned to W_l^p and W_l^n, respectively. Values that are neither positively nor negatively clustered are explicitly pruned. The clustering is performed based on the layer-wise threshold parameter $\pm\Delta_l$, calculated using the absolute maximum value of the real-valued weights and the hyperparameter t, as:

$$\Delta_l = t \cdot max(|\mathbf{w}_l|). \tag{5.2}$$

The sparsity within the quantized weight tensor can be controlled through the global hyperparameter t, where a higher threshold results into more sparsity. This representation is beneficial because it increases the model complexity through trainable bit representations which are optimized using gradient descent. At the same time, the hyperparameter t can be manually tuned in order to maximize

sparsity for a targeted accuracy or to tradeoff prediction performance with resource efficiency.

5.1.2 Quantizing Activations

RaS can efficiently implement non-uniform weight representations and benefits of sparsity, however, it relies on uniform representations for activations. Such representation are limited by integer and fixed-point formats that can be calculated using the integer ISA of general-purpose processors. Fixed-point formats are better suited for activations because they can be represented as probabilities in the interval of [0, 1] and, thus, would need scaling when implemented with integer representation. While fixed-point formats can easily express the probability interval by a variable length fractional part, they are usually not supported by general-purpose processors and can potentially cause a significant instruction overhead when computed with integer hardware.

The RaS algorithm can avoid such instruction overheads on integer hardware (further discussed in Section 5.2) and, therefore, enables the use of fixed-point formats. In order to quantize the pre-activations a_i, they need to be bound to the interval of [0, 1] as:

$$a_i^c = \begin{cases} 0 : \tilde{a}_i \leq 0 \\ \tilde{a}_i : 0 < \tilde{a}_i < 1 \\ 1 : \tilde{a}_i \geq 1 \end{cases} \tag{5.3}$$

which ultimately extends the ReLU functions by an additional clipping function. After passing the bounding function, the activations can be quantize to k-bit fixed-point formats $\widetilde{a}_i$ as:

$$\widetilde{a}_i = \frac{1}{2^k - 1} round((2^k - 1)a_i^c)), \tag{5.4}$$

where k represents the bit width which needs to be set statically. RaS mainly uses $k = 8$, because 8-bit representations are usually enough to avoid accuracy degradation during quantization. Furthermore, 8-bit formats can be excellently mapped to most general-purpose processors while lower-precision formats do not map well. However, the bit width of activations is in general configurable to any $k \geq 1$. Although there is also a large amount of sparsity within the activation tensors (induced by the activation function), RaS does not exploit this during

inference.

5.2 Inference

The main operation during inference of neural networks is to calculate scalar products of activation vectors $\mathbf{a}$ and weight vectors $\mathbf{w}$ with equal length N. For real-valued operands, this requires multiplying activations with their respective weights and accumulating the products into the resulting scalar c as:

$$c = \sum\nolimits_{i=1}^{N} a_i \cdot w_i, \quad a_i, w_i \in \mathbb{R} \quad \forall i. \tag{5.5}$$

Thus, the inference can be decomposed into a large number of MAC operations. The overall objective is to reduce the requirements of such MAC computations as well as the memory accesses to the operands.

5.2.1 Sparse and Integer Inference

The inference requirements can be reduced by applying pruning and incorporating sparse algorithms. For real-valued weights and activations that contain a certain sparsity level within the parameters, the scalar products can be calculated as:

$$c = \sum_{i \in \mathbf{i}_l} a_i \cdot w_i, \quad where \quad \mathbf{i}_l = \{i || w_i| > 0\}, \tag{5.6}$$

using float-point arithmetic. Vectorization is applicable to sparse algorithm by broadcasting w_i to all S operands along the SIMD lane and computing several $a_{i,j} ... a_{i,j+S}$ in parallel. Similar applies to parallelization: although there is the potential issue of load imbalance between threads, this does not result into inferior performance on general-purpose CPUs due to the small amount of parallel cores. While sparse algorithms are able to effectively reduce computations, the main drawback of purely sparse inference is the high number of memory accesses, because non-zero floating-point values as well as their indices need to be fetched from memory.

It is also viable to combine sparse algorithms with low-precision formats, in order to further reduce the amount of memory accesses. While it is practically infeasible to leverage sparsity using the bit-serial technique and, hence, use extreme forms of quantization, it is beneficial to use the rather conservative 8-bit

or 16-bit formats. Real numbers with fractional parts need to be represented as integers to take advantage of low-precision formats on general-purpose hardware, which is a trade off between precision and efficiency. The generalized fixed-point format is represented as $[Q_I.Q_F]$, where Q_I denotes the bits of the integer and Q_F denotes the bits of the fractional part of the number. Let the activations be represented by only an 8-bit fractional part (e.g. Q0.8) and weights be represented by only an 8-bit integer part (e.g. Q8.0), which results in computing the MACs as:

$$(Q16.16)Accu = (Q16.16)Accu + (Q8.8)((Q0.8)I \cdot (Q8.0)W), \tag{5.7}$$

$$(int32)Accu = (int32)Accu + (int16)((int8)I \cdot (int8)W). \tag{5.8}$$

The bit with of both operands need to be summed up $(Q(8+0).(0+8) = Q8.8)$ to guarantee no overflows during multiplication, resulting in 16-bit integer instructions. Next, the calculated $Q8.8$ product is added to the larger $Q16.16$ accumulator (avoiding overflows), which requires 32-bit integer instructions. Compared to single-precision floating point, this means an theoretical improvement of 4× for fetching operands from memory, 2× for multiplication and 1× for accumulation. However, the practical improvements are rather low, since floating-point MAC can be performed with one instruction, while Equation 5.7 requires two instructions.

5.2.2 RaS Inference

According to the weight quantization (Equation 5.1) , the weight tensor of a layer consists of real-valued elements $\{W_l^p, 0, W_l^n\}$. Based on this representation, positively and negatively weighted values can be treated independently and zero-weighted values can be skipped. As a consequence, the calculation for RaS can be reformulated and scalar products can be obtained as:

$$c = W_l^p \cdot \sum_{i \in \mathbf{i}_l^p} a_i + W_l^n \cdot \sum_{i \in \mathbf{i}_l^n} a_i, \quad where \tag{5.9}$$

$$\mathbf{i}_l^p = \{i | w_i = W_l^p\} \quad \text{and} \quad \mathbf{i}_l^n = \{i | w_i = W_l^n\}. \tag{5.10}$$

This results in only two real-valued multiplications per scalar product and reduces the major part of the computations to additions. Furthermore, only the indices

and activations need to be fetched from memory, while common sparse algorithms also require the weights. Parallelization and vectorization is also applicable to this algorithm by computing several a_i in parallel and vectorizing $a_{i,j}...a_{i,j+S}$ along the SIMD lane.

According to the activation quantization (Equation 5.3), the activations of a layer are represented by an 8-bit fractional fixed-point format. Combined with the accumulation technique of RaS (Equation 5.9), this enables to compute the MAC as:

$$(Q8.8)Accu = (Q8.8)Accu + (Q0.8)I, \tag{5.11}$$

$$(int16)Accu = (int16)Accu + (int8)I. \tag{5.12}$$

As can be seen, this results into a much more efficient calculation than the sparse technique: by avoiding the multiplication, the requirement of expanding the bit width is removed, which allows the use of simple 16-bit integer instructions for the major part of the computations. As reported in Table 5.1, these instructions are very fast (in comparison to floating-point MACs) while also being cheap in terms of energy. The overall benefits of the RaS technique can be summarized as:

Table 5.1 Cycles and energy per MAC and addition (ADD) operation. [4]

Instruction	Cycles (normalized)	Energy (pJ)
float32 MAC	8	4.60
int16 MAC	3	1.60
int16 ADD	1.5	0.05

i) Efficient calculation of fixed-point arithmetic on integer hardware by avoiding multiplications and bit-width expanding. ii) Virtually removing memory accesses to weights. iii) Efficiently leveraging sparsity and reduced-precision formats.

5.2.3 RaS Algorithm

RaS is a promising candidate for efficient inference on general-purpose processors, however, certain code optimizations that are implemented in BLAS libraries can not be applied. Such optimizations include memory tiling techniques (depending on the memory hierarchy of the processor) are not compatible with sparse memory accesses. Consequently, the RaS algorithm requires novel techniques for achieving

competitive performance to such libraries. This section describes the design of the algorithm with respect to the used quantization functions and its implications to the processors.

First, the original weight tensor $\mathbf{W}_l$ is transposed into $\mathbf{W}_l^T$ to increase data locality during runtime. This leads to a sparse ternary tensor with arbitrary positive and negative scaling factors (as illustrated in Equation 5.13).

$$\mathbf{W}_l^T = \begin{pmatrix} 0 & W_l^p & W_l^p & 0 & W_l^n \\ W_l^n & 0 & 0 & W_l^p & 0 \\ W_l^p & W_l^n & 0 & W_l^n & W_l^n \end{pmatrix} \tag{5.13}$$

The next step is to create an appropriate sparse format for the quantized parameters: positive and negative weighted parameters are split into two separate arrays and the zeros-weighted elements are removed. Then all indices i for which the respective element in the weight tensor $\mathbf{W}_l^T$ is either W_l^p or W_l^n are extracted and their indices are stored in $\mathbf{I}_l^p$ or $\mathbf{I}_l^n$. This separation of $\mathbf{W}_l^T$ into $\mathbf{I}_l^p$ and $\mathbf{I}_l^n$ is illustrated as:

$$\mathbf{I}_l^p = \begin{pmatrix} 1 & 2 & \\ 3 & & \\ 0 & & \end{pmatrix} \quad \text{and} \quad \mathbf{I}_l^n = \begin{pmatrix} 4 & & \\ 0 & & \\ 1 & 3 & 5 \end{pmatrix}. \tag{5.14}$$

Creating this sparse format is done after training and before deployment and, hence, does not cause additional processing overhead at inference.

After transforming the weight tensor into the sparse format, Algorithm 1 is used to compute the RaS scalar products. The algorithm requires the 8-bit fixed-point activation tensor $(int8)Input$, the 16-bit sparse lists $\mathbf{I}_l^p$ and $\mathbf{I}_l^n$, as well as both real-valued scaling factors W_l^p and W_l^n as inputs. There are two accumulator arrays $Accu_p$ and $Accu_n$ used to accumulate the 8-bit fixed-point inputs on the basis of the previously obtained indices. These accumulations are performed on the basis of Equation 5.11, using vectorized 16-bit integer addition. After accumulating the inputs, the results are cast to single-precision floating point, multiplied with the scaling coefficients W_l^p and W_l^n and finally accumulated to the result.

The algorithm is optimized for ARM processors, however, the same optimizations are also applicable to other processors. Parallelization is implemented

using OpenMP (in Operation 1) without the need of synchronization and the accumulations (in Operation 5 and 11) are vectorized using the ARM NEON processor extension.

Algorithm 1 RaS matrix-matrix multiplication algorithm. [4]

Input: $(int8)Input, (int16)\mathrm{I}_l^p, (float32)\mathrm{W}_l^p, (int16)\mathrm{I}_l^n, (float32)\mathrm{W}_l^n$

Output: $(float32)Output$

1: **for** $row := 0$ to $Input.rows()$ **do**
2: **for** $el := 0$ to $\mathrm{I}_l^p[row].elements()$ **do**
3: $index \leftarrow \mathrm{I}_l^p[row][el]$
4: **for** $col := 0$ to $Input.cols()$ **do**
5: $(int16)\mathrm{Accu}_p[col] \leftarrow (int16)\mathrm{Accu}_p[col] + (int8)Input[row][index]$
6: **end for**
7: **end for**
8: **for** $el := 0$ to $\mathrm{I}_l^n[row].elements()$ **do**
9: $index \leftarrow \mathrm{I}_l^n[row][el]$
10: **for** $col := 0$ to $Input.cols()$ **do**
11: $(int16)\mathrm{Accu}_n[col] \leftarrow (int16)\mathrm{Accu}_n[col] + (int8)Input[row][index]$
12: **end for**
13: **end for**
14: **for** $col := 0$ to $Input.cols()$ **do**
15: $(float32)\mathrm{Res}_p \leftarrow castFixedPoint16ToFloat32((int16)\mathrm{Accu}_p[col])$
16: $(float32)\mathrm{Res}_n \leftarrow castFixedPoint16ToFloat32((int16)\mathrm{Accu}_n[col])$
17: $Output[row][col] \leftarrow (float32)\mathrm{W}_l^p \cdot \mathrm{Res}_p + (float32)\mathrm{W}_l^n \cdot \mathrm{Res}_n$
18: **end for**
19: **end for**

5.3 Compression

Storage requirements of DNNs are one of the most critical factors when deploying them onto resource-constrained environments. The vast majority of storage is produced by the weights of a model, while biases and other parameters - such as batch-normalization coefficients - only contribute a negligible amount. One can easily imagine that an embedded systems in a real-world scenario does not only employ a single, but rather several models for different use cases. Transmitting

these models for each deployment is infeasible, since it results in high latency and it requires online connectivity, which may be not given due to environmental or security reasons.

Based on the quantization function of the weights (Equation 5.1), only two bits are required to store each weight, which already exploits a compression rate of 16× compared to the single-precision counterpart. However, the ternary weights can also contain a large amount of sparsity, indicating an even further potential for compression.

In order to leverage the sparsity in combination with ternary weights, the transposed weight tensor $\mathbf{W}_l^T$ is flattened, the signs are stored together with their indices and Huffman coding is applied. Both indices and signs are organized as vectors ($\mathbf{i}_l$ respectively $\mathbf{s}_l$). First, the indices and signs of the scaling coefficients are determined by evaluating all non-zero elements in the weight matrix ($\mathbf{i}_l = \{i|w_l^i \neq 0\}$). For the non-zero signs of vector $\mathbf{s}_l$, only a single bit per element is required to distinguish between positive and negative scaled weights. After extracting the non-zero indices into vector $\mathbf{i}_l$, the distance vector $\mathbf{d}_l$ is calculated, which is based on the distances between consecutive elements of the previously obtained index vector ($\mathbf{d}_l^j = \mathbf{i}_l^{j-1} - \mathbf{i}_l^j$, with $\mathbf{i}_l^{-1} = 0$). Obtaining $\mathbf{s}_l$ and $\mathbf{d}_l$ from $\mathbf{W}_l^T$ is illustrated in Eq. 5.15, 16, 17 and 18.

$$\mathbf{W}_l^T = \begin{pmatrix} 0 & W_l^p & W_l^p & 0 & W_l^n \\ W_l^n & 0 & 0 & W_l^p & 0 \\ W_l^p & W_l^n & 0 & W_l^n & W_l^n \end{pmatrix} \tag{5.15}$$

$$\mathbf{s}_l = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \tag{5.16}$$

$$\mathbf{i}_l = \begin{pmatrix} 1 & 2 & 4 & 5 & 8 & 10 & 11 & 13 & 14 \end{pmatrix} \tag{5.17}$$

$$\mathbf{d}_l = \begin{pmatrix} 1 & 1 & 2 & 1 & 3 & 2 & 1 & 2 & 1 \end{pmatrix} \tag{5.18}$$

Evaluating the distance vector ultimately reduces the amount of possible values and increases the frequency of appearance of the obtained values. In order to leverages the frequency of appearance within the distance vector $\mathbf{d}_l$, Huffman coding is applied. This encoding technique uses fewer bits to encode values with a high and more bits to encode values with lower frequency of appearance. A single codebook is used that contains the codes for all layers, in order to reduce the search space and decoding is simply done by looking up the values in the codebook.

5.4 Evaluation

The efficiency of RaS is evaluated in terms of test accuracy, memory footprint and inference rate in Frames Per Second (FPS) on an ARM Cortex-A57 processor. This evaluation is done on three different DNNs and datasets: a simple ConvNet on SVHN, ResNet-44 on CIFAR-10 and AlexNet on ImageNet.

The proposed RaS technique is compared to single-precision floating point (*baseline*), binarization (*BNN*) and an 8-bit integer implementation (*Int8*). The baseline implementation relies on the GEMM operator of the Eigen library for benchmarking the throughput. BNN quantization is performed using DoReFa-Net and the extended version of the Eigen library (as described in Section 4.1) for benchmarking the throughput. Google's gemmlowp library [1] is used for benchmarking 8-bit integer inference, while the accuracy is not reproduces because no prediction degradation is assumed for such representations.

The sensitive first and last layer of the architectures are not quantized (as commonly practised) in order to avoid sacrificing accuracy. Furthermore, all GEMM operations are vectorized and parallelized in order to guarantee a fair comparison. Last, all reported results are based on equal models, including identical hyperparameters and epochs for training.

5.4.1 SVHN

Results for the four different implementations on the SVHN task are summarized in Table 5.2 where the single-precision baseline model achieves 97.5% accuracy with a classification rate of 258 FPS and a memory footprint of 8.3 MB. Binarization improves the baseline throughput by 5.5× and reduces storage requirements by 27.8× while only sacrificing 0.5% classification accuracy. Hence, SVHN is a prime example where binarization achieves excellent performance improvement. Using 8-bit integer for inference only improves the throughput by 1.4× and storage by 4.0×. This indicates that there is much more redundancy in the model, however, 8-bit quantization is not able to leverage it. Similar to binarization, RaS is able to create 89% sparsity within the weight tensors and accelerate throughput by 5.2× and memory by 43.3× while achieving baseline accuracy.

[1] https://github.com/google/gemmlowp

Table 5.2 Results for ConvNet on the SVHN dataset. [4]

	Baseline	BNN	Int8	RaS
Accuracy	97.5%	97.0%	–	97.5%
Sparsity	0%	0%	0%	89%
Inference Rate	258 FPS	1,409 FPS	368 FPS	1,337 FPS
Memory Footprint	8,321kB	299kB	2,080kB	192kB

Consequently, RaS is able to adapt to this rather simple task and outperforms all other implementations.

5.4.2 CIFAR-10

Next, the results on the more complex CIFAR-10 tasks are evaluated using the popular ResNet architecture with 44 layers (see Table 5.3). As can be seen, binarization is able to accelerate computation by a factor of 7.7× and compresses memory by 32.0×. These impressive improvements come at the cost of an 5.0% accuracy drop, which highlights the drawbacks of binarization on complex tasks. Again, 8-bit integer inference achieves a solid improvement of

Table 5.3 Results for ResNet-44 on CIFAR-10 dataset.[4]

	Baseline	BNN	Int8	RaS
Accuracy	92.6%	87.6%	–	92.4%
Sparsity	0%	0%	0%	58%
Inference Rate	69 FPS	532 FPS	100 FPS	191 FPS
Memory Footprint	2,622kB	82kB	655kB	117 kB

1.5× higher throughput and 4.0× less storage. On the contrary, RaS can adapt to the task and model and improves throughput by 2.8× and memory by 22.4× by leveraging 58% sparsity. This indicates that RaS also performs reasonable well under rather low sparsity levels. In summary, binarization results in a high accuracy degradation and int8 into mediocre improvements, while RaS achieves superior performance by compromising efficiency and accuracy.

5.4.3 ImageNet

Last, all four implementations are evaluated on the competitive and large-scale image task ImageNet in Table 5.4. Here, binarization accelerates computations

by 5.5× and reduces storage by 10.2×. The accuracy degradation for binarized networks is 10.8% and 21.9% for Top1 and Top5 accuracy, respectively, indicating that such aggressive quantization techniques fail on large-scale image tasks. The

Table 5.4 Results for AlexNet on ImageNet dataset.[4]

	Baseline	BNN	Int8	RaS
Top-1 Accuracy	56.2%	45.4%	–	56.4%
Top-5 Accuracy	78.3%	56.4%	–	79.0%
Sparsity	0%	0%	0%	63%
Inference Rate	4 FPS	22 FPS	7 FPS	8 FPS
Memory Footprint	244MB	24MB	61MB	25MB

rather conservative 8-bit integer representation performs much better on such complex tasks and achieves 1.8× and 4.0× improvement in terms of classification ratio and memory, respectively. Similar to this is the performance of RaS, which accelerates inference by 2.0× and reduces memory by 9.8× at 63% sparsity.

5.4.4 Trading Accuracy with Efficiency

The main objective of compression techniques is to remove as much redundancy as possible while maintaining prediction performance. Experiments on SVHN, CIFAR-10 and ImageNet show that neither extreme quantization through binarization nor conservative representation through 8-bit integer formats are able to adapt to this objective. However, RaS shows excellent adapting properties in this context.

Deploying DNNs in resource-constrained systems or under real-time requirements enforces certain demands on memory footprint, latency and inference rate. In order to meet these demands, it is more efficient to train a larger model and accept a trade-off between accuracy and efficiency. For instance, an increase of the sparsity threshold from $t = 0.30$ to $t = 0.35$ for the RaS technique on the SVHN dataset results in only 0.7% accuracy degradation, while the achieved sparsity increases from 89% to 96%. This sparsity increase reduces the memory footprint to 95kB and increases the inference rate to 1,538 FPS. Table 5.5 and 5.6 report the model metrics for a varying sparsity threshold t for CIFAR-10 and ImageNet, respectively.

Applied to the CIFAR-10 task, an accuracy loss of 1.5% and 2.5% can accelerate inference to 3.7× and 4.3×, while reducing storage requirements

Table 5.5 Results for RaS using ResNet-44 on CIFAR-10 for a varying threshold-parameter t. [4]

	$t = 0.25$	$t = 0.30$	$t = 0.35$
Accuracy	92.4%	91.1%	90.1
Sparsity	58%	63%	71%
Inference Rate	191 FPS	255 FPS	293 FPS
Memory Footprint	117 kB	108 kB	90 kB

to 24.3× and 29.1×, respectively. Similar can be observed on the ImageNet task where 0.8% and 1.3% less Top-5 accuracy enables 4.3× and 5.0× faster computation at 11.1× and 12.8× reduced storage.

Table 5.6 Results for AlexNet on ImageNet for a varying threshold-parameter t. [4]

	$t = 0.05$	$t = 0.10$	$t = 0.15$
Top-1 Accuracy	56.4%	54.7%	53.7%
Top-5 Accuracy	79.0%	77.5%	77.0%
Sparsity	63%	78%	88%
Inference Rate	8 FPS	17 FPS	20 FPS
Memory Footprint	25MB	22MB	19MB

The results obtained in this section highlight the sensitivity of the models on different tasks to sparsity: while all evaluated architectures allow a certain amount of sparsity (typically $> 60\%$) without sacrificing prediction performance, they rapidly loose accuracy after reaching their critical sparsity level. At the same time, RaS is able to leverage even small increases of sparsity and, therefore, allows excellent trade-offs between accuracy and efficiency.

5.4.5 Scaling BNNs

Binarization is highly effective in terms of inference rate and memory requirements, but can cause severe accuracy degradation if the task is too complex. One could argue that the accuracy loss can be compensated by either making the neural network wider (increasing the amount of neurons per layer) or deeper (increasing the amount of layers), which is basically a tradeoff between training and inference time.

In this experiment, the ResNet architecture on CIFAR-10 is scaled by deepening the binarized model until it roughly reaches full-precision accuracy (ResNet-44

baseline). The results show that binarization requires an increase from 44 to 272 layers in order to achieve 91.2% test accuracy. The resulting model is still much smaller than the baseline model, however, it is 4.7× larger than a RaS model with similar accuracy. In addition, the scaling of BNNs requires 6.5× more training time in comparison to RaS.

5.4.6 Comparing to Deep Compression

RaS is not the first compression technique that leverages sparsity in the weight tensor while quantizing weights and activations differently. Deep Compression [51] is the most popular compression technique in this context: it quantizes the weights of convolution layers to 8 bit, the dense layers to 5 bit (including an explicit pruning function) and the activations to 16 bit. The indices to non-zero weights are further compressed by Huffman coding. Deep Compression is compared to RaS in Table 5.7 on the example of AlexNet on the ImageNet task.

Table 5.7 Comparison of Deep Compression and RaS (memory footprint and sparsity refer to AlexNet on ImageNet). [4]

	Deep Compression	RaS
Top-1 Accuracy	57.2%	56.4%
Top-5 Accuracy	80.3%	79.0%
Memory Footprint (all layers)	6.9 MB	25.2 MB
Memory Footprint (hidden layers only)	5.9 MB	8.7 MB
Compression Ratio (all layers)	97.1%	89.7%
Compression Ratio (hidden layers only)	97.4%	96.2%
Sparsity (all layers)	89%	63%
Sparsity (hidden layers only)	90%	68%
Energy costs per PE (using data from)	153.6pJ	15.4pJ
CLB LUTs per PE	765	64
Maximum frequency of PEs	384MHz	400MHz

As can be seen, Deep Compression achieves 7.4% better compression ratio and 26% more sparsity than RaS, which is mainly because RaS does not quantize the first and last layers, and the output layer of AlexNet alone has a size of 16 MB. When comparing only the quantized layers, Deep Compression achieves only 1.2% better compression ratio. Still, Deep Compression achieves about 22% more sparsity which directly translates into fewer operations, but relies on 5-/8-bit multiplications with 16-bit activations. An FPGA synthesis report

(Vivado design suite) indicates that for one Processing Element (PE) consisting of 8 channels and using 8×8 inputs, a MAC unit requires roughly 10 times more Configurable Logic Blocks (CLBs) than an accumulator, as it is required for RaS.

Furthermore, Deep Compression is designed to be employed on the Efficient Inference Engine (EIE) [79], a specialized ASIC for performing inference tasks on compressed neural networks. However, it does not enable better efficiency on general-purpose processors (such as ARM CPUs) while RaS is specifically designed to be employed on these forms of processors.

5.5 N-Ary Quantization

The previous sections showed that the RaS technique is able to compression and accelerate inference for a variety of models and tasks. As discussed in Section 5.1, RaS leverages the non-uniform ternary quantization TTQ [32] for weights and a simple fixed-point representation for activations. While this discretization is able to generate excellent efficiency on rather simpler tasks or complex tasks on highly over-parameterized architectures (such as AlexNet or VGG), it results in sever performance degradation on large-scale images (i.e. ImageNet) on modern architectures (i.e. ResNet or Inception) as well as recurrent neural networks.

This section addresses the issue of accuracy degradation by introducing *N-Ary Quantization*, a scalable non-uniform quantization technique for weights, that is based on trainable scaling factors in combination with nested-means clustering. The main idea of this clustering technique is to split the weight distribution iteratively into multiple quantization intervals until a pre-defined discretization level is reached. By adopting the concept of TTQ [32], all weights within a certain quantization interval are assigned the same trainable scaling factor. Activations are quantized following a linear discretization technique that takes the statistical properties of batch normalization into account and is inline with the RaS inference algorithm.

5.5.1 Weight Quantization

The n-ary quantization technique follows the same basic concept as previously discussed techniques, where full-precision weights are maintained for training and quantized during forward propagation. The gradient of the full-precision weights

is approximated by backpropagating through quantization functions using STE. However, n-ary differs from previous techniques by introducing multiple scaling factors, novel weight representations and scalable nested-means clustering

Scaling Factors

TTQ [32] uses two independent and trainable non-uniform scaling factors α_+^l and α_-^l per layer that are determined by gradient descent and represent the non-zero weights. This adjusts the scaling factors so as to minimize the given loss function and, thus, is more accurate than other proposed scaling factors (i.e. mean of absolute floating-point weights). At the same time the model capacity is increased significantly due to non-uniform scaling. As a consequence, the n-ary quantization adopts these gradient-based scaling factors and extends the ternary case to arbitrary intervals. More formally: let $\delta_{-K_n}^l < \ldots < \delta_{-1}^l < 0 < \delta_{+1}^l < \ldots < \delta_{+K_p}^l$ be a set of interval thresholds that partition the real numbers into intervals

$$\mathbf{\Delta}_{-K_n}^l = \left(-\infty, -\delta_{-K_n}^l\right),\ \mathbf{\Delta}_{-i}^l = \left[\delta_{-i-1}^l, \delta_{-i}^l\right), \tag{5.19}$$

$$\mathbf{\Delta}_{+i}^l = \left[\delta_{+i}^l, \delta_{+i+1}^l\right),\ \mathbf{\Delta}_{+K_p}^l = \left[\delta_{+K_p}^l, \infty\right). \tag{5.20}$$

To each interval $\mathbf{\Delta}_i^l$, a trainable scaling factor α_i^l is assigned that is used to represent the weights as $w^q = \alpha_i^l \Leftrightarrow w \in \mathbf{\Delta}_i^l$. The scaling factors α_i^l are updated during training using gradients computed as

$$\frac{\partial E}{\partial \alpha_i^l} = \sum_{w^l \in \mathbf{\Delta}_i^l} \frac{\partial E}{\partial \left(w^l\right)^q}, \tag{5.21}$$

where E denotes the loss function and $\left(w^l\right)^q$ denotes the quantized weights. A fixed scaling factor $\alpha_0^l = 0$ is assigned to the sparse interval, which is defined as $\mathbf{\Delta}_0^l = [\delta_{-1}^l, \delta_{+1}^l)$ and is not updated during gradient descent.

N-Ary Representations

Non-uniform weight quantization enables various explorations of interesting weight representations, beyond binary and ternary forms. Weight distributions tend to be gaussian and symmetric around zero (see Fig. 5.1) and, hence, well approximating weight representations also exhibit a symmetry around zero. Table 5.8 summarizes several of such representations with a focus to be inline with the

RaS inference technique. Hence, only representation with sparsity potential are chosen, but arbitrary other forms are possible.

Table 5.8 Different non-uniform n-ary weight representations.

Notation	Representation	Bit width
Binary	$\{\alpha_{-0}, \alpha_{+0}\}$	1
Ternary	$\{\alpha_{-1}, 0, \alpha_{+1}\}$	2
Quaternary+	$\{\alpha_{-1}, 0, \alpha_{+1}, \alpha_{+2}\}$	2
Quaternary-	$\{\alpha_{-2}, \alpha_{-1}, 0, \alpha_{+1}\}$	2
Quinary	$\{\alpha_{-2}, \alpha_{-1}, 0, \alpha_{+1}, \alpha_{+2}\}$	3

A non-uniform binary representation might be beneficial for gaining accuracy while achieving high compression rates (32×) and, hence, enabling low storage costs. However, it is not suited for inference accelerations, since it violates the requirements for bit-serial calculations and it does not feature sparsity, which is a fundamental issue for RaS. As seen in the previous sections, non-uniform ternary representation offer both, low storage and high throughput potential.

Whereas these representations are well studied in related literature, other n-ary weight representations are possible and facilitate similar compression and inference levels while improving model capacity. For instance, quaternary weights can also be encoded with only two bits, but introduce either an additional positive (quaternary+) or negative (quaternary-) scaling factor, respectively. Quinary weights extend ternary weights by one positive and one negative value, but are still encoded with only two bits in a sparse format.

Nested-Means Clustering

Trainable scaling factors and appropriate weight representations are vital for prediction performance, however, the single most important factor for an optimal approximation is weight clustering, which is required to partition a set of weights that are later represented by a single discrete value per cluster (see Section 5.5.1). Such a clustering can be implemented either statically (once before retraining) or dynamically (repeatedly during training) by calculating thresholds δ, which represent the boundaries of the respective cluster.

The static variant has the advantage of allowing iterative clustering algorithms to be applied (e.g. k-means clustering), that are able to find optimal solution

for the cluster assignment. The clustering algorithm is usually applied to pre-trained real-valued parameters and subsequently fine tuned after quantization in an additional training step. As a consequence, the optimal solution found by an iterative algorithm is likely to become non-optimal during the following fine tuning process. On the contrary, applying an iterative clustering approach repeatedly during training is practically infeasible, since it causes a dramatic increase in training time.

A practical useful clustering solution is to calculate cluster thresholds during training based on the statistics of the underlying real-valued weight distribution. For instance, TTQ [32] uses the maximum absolute value (see Equation 5.2) to distinguish among positive, negative or zero quantized weights. The training time is virtually unaffected by this rather simple calculation. However, having an additional hyperparameter for each cluster renders the mandatory hand tuning infeasible for multiple clusters (i.e. quaternary or quinary). Furthermore, the sensitivity to the maximum value results in aggressive threshold changes caused by weight updates.

In order to overcome these issues, a symmetric nested-means clustering algorithm is applied for assigning full-precision weights to a set of quantization clusters. First, the weights are split into a positive and a negative cluster ($\mathbf{I}^l_{+1}$ and $\mathbf{I}^l_{-1}$). Then, these clusters are further divided at their arithmetic means δ^l_{+i} and δ^l_{-i} into two subclusters . Each cluster obtains an inner cluster and an outer cluster containing the tail of the distribution. The subclusters containing the tail of the distribution ($\mathbf{I}^l_{+i+1}$ and $\mathbf{I}^l_{-i-1}$) are repeatedly divided at their arithmetic means until the targeted number of quantization intervals is reached. More formally, nested-means clustering iteratively computes

$$\delta^l_{+i} = \frac{1}{|\mathbf{I}^l_{+i}|} \sum_{j \in \mathbf{I}^l_{+i}} w^l_j \quad \text{and} \quad \delta^l_{-i} = \frac{1}{|\mathbf{I}^l_{-i}|} \sum_{j \in \mathbf{I}^l_{-i}} w^l_j \tag{5.22}$$

$$\mathbf{I}^l_{+i+1} = \{j | w^l_j \geq \delta^l_{+i}\} \quad \text{and} \quad \mathbf{I}^l_{-i-1} = \{j | w^l_j < \delta^l_{-i}\}, \tag{5.23}$$

starting with $\mathbf{I}^l_{+1} = \{j | w^l_j \geq 0\}$ and $\mathbf{I}^l_{-1} = \{j | w^l_j < 0\}$. The whole clustering process is shown in Fig. 5.1 on the example of seven quantization clusters.

This clustering technique is beneficial in the context of RaS for several reasons: nested-means clustering dynamically defines the cluster thresholds in a way that the cluster intervals approximate the underlying weight distributions well. Next,

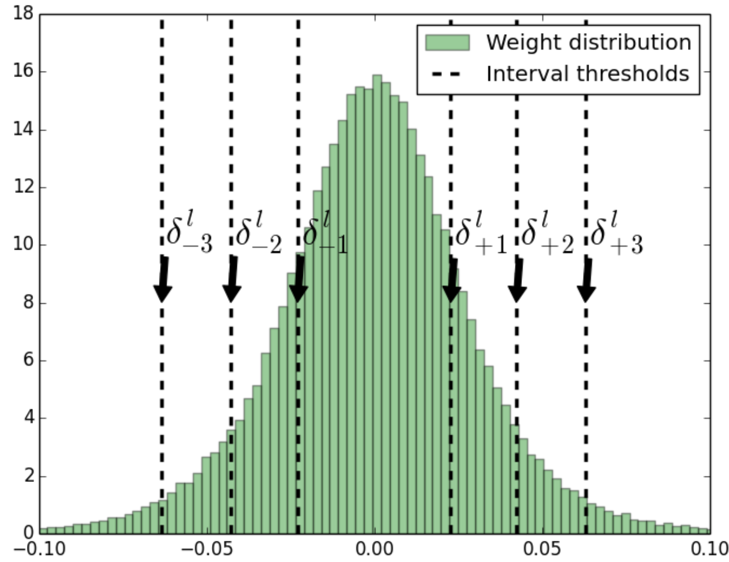


Fig. 5.1 Nested-means intervals of a trained ResNet layer.

the mean is less sensitive to weight updates than the maximum absolute weight value [32], allowing for more stability during training and better convergence. Also, nested-means clustering is free of hyperparameters and requires only one arithmetic mean per cluster, which is computationally efficient. Furthermore, the nested-means clustering is highly beneficial when increasing the pruning ratio: the inner cluster threshold can be extended with a configurable hyperparameter to $t \cdot \delta^l_{+1}$ and $t \cdot \delta^l_{-1}$, enabling to control the amount of sparsity within the model. The iteratively computed nested-means clustering allows that all other cluster thresholds adapt to increasing or decreasing t and, therefore, offers a high amount of flexibility when tuning the networks towards higher efficiency or accuracy.

5.5.2 Activation clipping

The activation quantization in Section 5.1.2 approximates the ReLU function through a clipping function, in order to bound the inputs to a layer into a predefined interval. This is necessary for the subsequently applied linear quantization function, since the ReLU function produces unbounded outputs. For simplicity, the upper bound of this clipping function is set to a fixed value of one, which showed empirically good performance. However, experiments on more recent architectures and large-scale image tasks indicate that this upper bound causes

accuracy degradation.

This section reasons about activation distributions and derives an appropriate clipping interval. First, it is necessary to modify the clipping function of Equation 5.3 to a more generic interval, which can be defined as:

$$a_i^c = \begin{cases} 0 : \tilde{a}_i \leq 0 \\ \tilde{a}_i : 0 < \tilde{a}_i < \gamma \\ \gamma : \tilde{a}_i \geq \gamma \end{cases}, \tag{5.24}$$

where γ denotes the hyperparameter for the upper bound. Consequently, the linear quantization functions needs to be also modified in order to adapt to the generic interval $[0, \gamma]$, which can be calculated as:

$$Q_a(x) = \underbrace{\frac{\gamma}{2^k - 1}}_{\text{scale}} \cdot \underbrace{\text{round}\left(\frac{2^k - 1}{\gamma} Q_c(x)\right)}_{\text{k-bit integer}}. \tag{5.25}$$

The selection of the clipping parameter γ ultimately results in a tradeoff between zero gradients and quantization errors: small values of γ produce zero gradients in the clipped interval $[\gamma, \infty]$. On the other side, large values of γ result in a large interval $[0, \gamma]$ that needs to be quantized. This is problematic if only a few bits are used as quantization errors might become large.

In order to define an appropriate clipping interval, the observation is used that pre-activations tend to have a Gaussian distribution [9]. In a Gaussian distribution, most values lie within a rather small range and there are only a few outliers that yield a high absolute range. For instance, 99.7% of the values lie within three standard deviations σ of the mean μ [80]. This empirical rule is a good approximation to filter out outliers and define the clipping interval as $\gamma = \mu + 3\sigma$.

This approach approximates the ReLU function well but suffers from the drawback that μ and σ need to be repeatedly calculated during training. In recent years, batch normalization [9] became a standard tool to accelerate convergence of state-of-the-art DNNs. Batch normalization transforms individual pre-activations to approximately have zero mean and unit variance across all data samples. Cai et al. [36] experimentally showed that the pre-activation distribution after batch normalization are all close to a Gaussian with zero mean and unit variance.

Therefore, a fixed clipping parameter $\gamma = 3$ is selected as it results in a small quantization interval $[0, \gamma]$ while also keeping the number of clipped activations $x > \gamma$ small.

5.5.3 Evaluation

The evaluation of the n-ary quantization technique, including the novel nested-means clustering as well as activation clipping, is evaluated on the large-scale image classification task ImageNet [81]. ResNet [14] and Inception [9] architectures are used as representatives for state-of-the-art architectures. The quantized networks are initialized with pre-trained real-valued parameters. All convolution and fully-connected layers are quantized except the input and output layers, to avoid accuracy degradation.

Weight Quantization

The n-ary weight quantization technique is evaluated on ternary, quaternary- and quinary representations and summarized in Table 5.9. Additionally, the results for TTQ [32] are reported for comparison.

As shown in Section 5.4.3, TTQ achieves floating-point baseline accuracy on ImageNet using the AlexNet architecture. However, there is an accuracy drop of 3.4% for TTQ on ResNet18 (in comparison to the baseline model), which shows the impact of quantization on modern architectures. Similar applies to the Inception architecture, where the accuracy degradation of TTQ is 6.1%. N-ary quantization improve the accuracy of ternary quantization (in comparison to TTQ) by 1.2% and 2.6% for ResNet18 and Inception, respectively, by using the adaptive and more stable nested-means clustering. Still, there is an accuracy gap of 2.2% and 3.5% of ternary to the baseline model.

The scalability potential of the n-ary technique can be used to reduce this accuracy gap by trading efficiency with prediction performance. For the ResNet18 model, quaternary and quinary can reduce the gap to 1.2% and 0.7% and for the Inception model to 2.3% and 1.4%, respectively. Furthermore, all representation show a stable learning behaviour throughout the training process, which can be seen on the training curves in Figure 5.2a. The sparsity level of all models are similar to the experiments using the AlexNet architecture, indicating similar

Table 5.9 Validation accuracy (Top1, Top5) of ResNet18 and Inception-BN on ImageNet for 2-bit weights.

	Weights	Activations	Sparsity	Top1	Top5
ResNet-18	bits	bits	%	%	%
Baseline	32	32	0	70.4	89.5
TTQ	2	32	–	67.0	87.3
Ternary	2	32	61	68.2	88.0
Quaternary-	2	32	57	69.2	88.7
Quinary	3	32	53	69.7	89.0
Inception-BN	bits	bits	%	%	%
Baseline	32	32	0	73.1	91.4
TTQ	2	32	58	67.0	87.3
Ternary	2	32	58	69.6	89.1
Quaternary-	2	32	52	70.8	89.8
Quinary	3	32	54	71.7	90.3

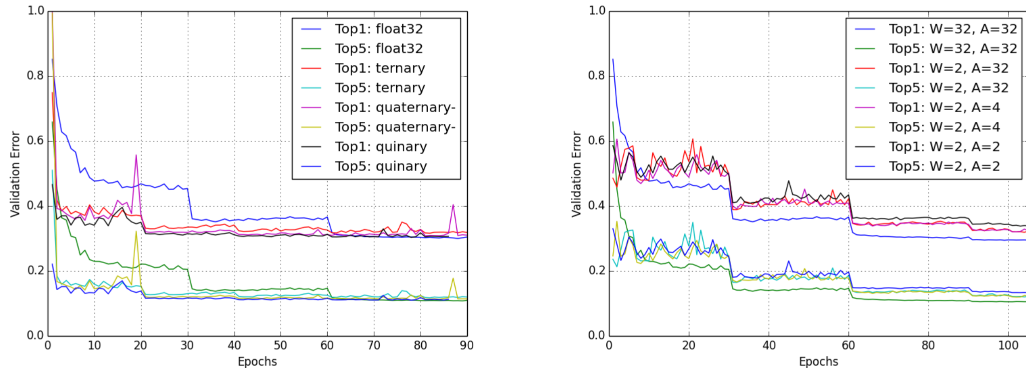
compression and inference improvements, however, significantly higher absolute accuracy.

Impact of Nested-Mean Clustering

Nested-means clustering naturally defines the cluster thresholds in a way that the cluster intervals become smaller for larger weights (see Figure 5.1). The results on ResNet and ImageNet indicate that such properties achieve high prediction performance. However, it is unclear whether this observation is true or the models are resilient to that. This section validates the effectiveness of the nested-mean clustering by comparing it to quantile clustering.

Assuming that the weights are approximately Gaussian distributed, the cluster thresholds can be computed so that each cluster approximately contains a pre-specified amount of weights. Let $\Phi^{-1}(p)$ be the quantile function of the standard Gaussian distribution with zero mean and unit variance. Given a vector $\boldsymbol{p} = (p_1, \dots, p_L)$ of pre-specified cluster probabilities that sum to one, the thresholds are computed as $\delta_i^l = \mu_w^l + \sigma_w^l \Phi^{-1}(\sum_{j \leq i} p_j)$, where μ_w^l and σ_w^l are the mean and the standard deviation of the weights in layer l, respectively.

Table 5.10 summarizes the accuracy using quinary weights of both nested-means and quantile clustering for several cluster sizes $\boldsymbol{p}$. First, cluster sizes are defined equally before the cluster size of smaller weights are incrementally



(a) For varying weight representations. (b) For varying activation representations.

Fig. 5.2 Training curves of validation error (Top1, Top5) of ResNet-18 on ImageNet.

Table 5.10 Validation accuracy (Top1) using quinary weights of nested-means clustering and quantile-clustering for several quantiles for ResNet18 on ImageNet.

Clustering Method					Top1 [%]
Nested-Mean					69.7
$p(\alpha_{-2})$	$p(\alpha_{-1})$	$p(0)$	$p(\alpha_{+1})$	$p(\alpha_{+2})$	
20%	20%	20%	20%	20%	68.8
11%	22%	33%	22%	11%	69.3
8%	17%	50%	17%	8%	69.4
6%	11%	66%	11%	6%	69.2

increased while, at the same time, the cluster size of larger weights are decreased. The accuracy improves if larger clusters are assigned to small weights and smaller clusters to large weights, which consequently validates the hypothesis.

Activation quantization

The last experiment evaluates the activation quantization using ResNet-18 on ImageNet. The clipped ReLU and $\gamma = 3$ is used for quantized activations and compared to the ReLU without clipping and quantization for 32-bit activations. Table 5.11 reports the validation accuracy for several activation bit widths in combination with ternary quantization for weights.

As can be seen, the clipping interval $[0, 3]$ for activation, that is motivated by the statistical attributes of batch normalization, virtually does not affect

Table 5.11 Validation accuracy (Top1,Top5) and increase in training time for different activation bit-width of ResNet-18 on ImageNet.

Weights	Activations	Top1	Top5
Ternary	32	68.2	88.0
Ternary	8	68.1	88.0
Ternary	4	68.1	88.1
Ternary	2	66.2	86.7

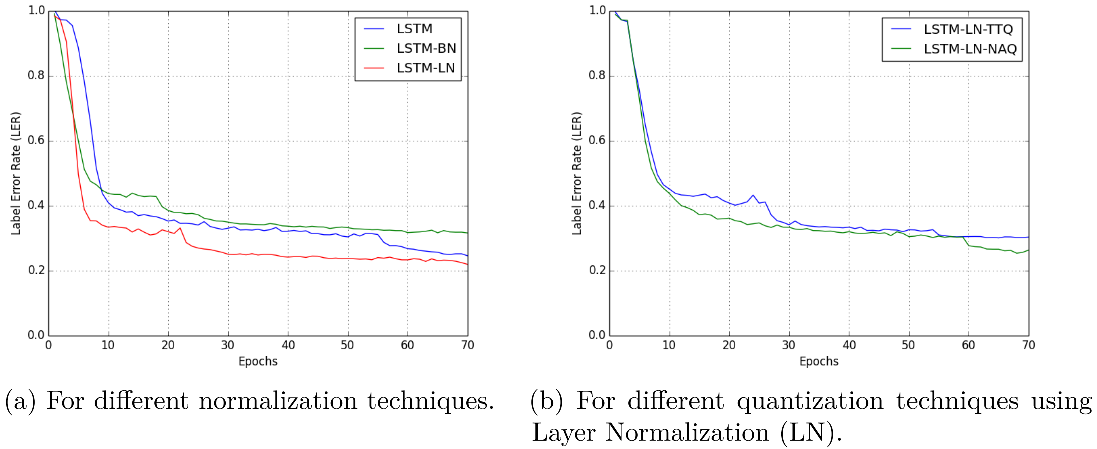
prediction performance for bit width larger or equal to 4 bit. Only a very aggressive quantization down to 2 bit results in a 2.0% accuracy drop. Last, all evaluated activation bit widths result in a stable learning behaviour, as can be seen on the training curves in Figure 5.2b.

Quantizing LSTMs

The previous sections evaluated several quantization techniques on convolution and fully-connected layers on image recognition tasks without considering recurrent neural networks. In this section, quantization experiments are performed on LSTM layers using a speech recognition task and the implications of such recurrent architectures are studied under low-bit considerations. This experiment is performed on a real-world temporal classification problem: phonetic labelling on the TIMIT speech corpus. The task is to annotate the utterances in the TIMIT test set with the phoneme sequences that gives the lowest possible Label Error Rate (LER). Model and experimental setup follow the initial implementation of Graves et al. [82].

Applying weight quantization to LSTMs in the original setup of Graves et al. [82] leads to *Not a Number* exceptions and, hence, does not converge. The main problem here are exploding gradients when the underlying real-valued weights of the model become very large. In order to resolve these exploding gradients, it is necessary to apply a normalization technique which can stable each value to reduce the gradient problem.

Batch Normalization (BN) [9] is the most popular normalization technique due to its excellent performance on convolution and fully-connected layers. BN is also applicable to LSTMs and, furthermore, resolves exploding gradients when quantization is applied. However, the experiments on the TIMIT task indicate,



(a) For different normalization techniques.

(b) For different quantization techniques using Layer Normalization (LN).

Fig. 5.3 Training curves of Label Error Rate (LER) using LSTMs on TIMIT.

Table 5.12 LER on TIMIT for various normalization, quantization techniques and Weights (W) representations.

Method	W	Train	Validation
LSTM	32	24.6	30.0
LSTM+TTQ	2	NaN	NaN
LSTM+NAQ	2	NaN	NaN
LSTM+BN	32	31.6	34.8
LSTM+LN	32	21.9	26.6
LSTM+LN+TTQ	2	30.1	31.4
LSTM+LN+NAQ	2	25.4	28.2
LSTM+LN+NAQ	1	28.4	30.0

that BN leads to severe performance degradation. As can be seen in Figure 5.3a, BN converges similarly fast than the unnormalized model, but results in 4.8% less final accuracy. Another normalization technique is Layer Normalization (LN) [83], which has proven successful on recurrent neural networks as well. LN converges faster than BN and the unnormalized model and also outperforms the baseline by 3.4% accuracy.

Figure 5.3b shows the learning curves of the model using LN in combination with TTQ and n-ary quantization. As can be seen, both quantization techniques converge due to the applied normalization, but n-ary quantization outperforms TTQ. The final validation accuracy of the n-ary technique is 3.2% more accurate than TTQ while both are using ternary representation. Surprisingly, binary quantization also outperforms TTQ by 1.4% even though it uses only half the

Table 5.13 LER on TIMIT for varying sparsity levels..

t	SPARSITY	TRAIN	VALIDATION
1.0	62%	25.4	28.2
1.4	75%	26.9	28.9
1.6	80%	27.2	29.4
1.8	84%	27.4	29.2
1.8	87%	34.3	34.3

bits for the weights of the LSTMs.

Last, Table 5.13 reports accuracy for varying sparsity levels of the model by increasing the threshold parameter t. The results indicate a similar sparsity level than convolutions and, consequently, a similar acceleration potential using RaS inference. Furthermore, n-ary is able to create more than 80% sparsity within the ternary weights without significantly reducing prediction accuracy. This ultimately shows the compression potential of the n-ary technique, since it is applicable and efficient on convolution, fully-connected as well as recurrent layers.

5.6 Summary

Approximating weights using non-uniform representations achieves impressive theoretical improvements and does not result into severe accuracy degradation. However, such representations are not supported in common software stacks and, hence, do not result into inference acceleration by default. The proposed RaS technique is able to adapt to this representation and leverages extreme low-precision formats as well as sparsity while being inline with hardware requirements of modern CPUs. The key insights of this sections are:

- Extreme forms of quantization (as discussed in Section 4.2) fail to achieve single-precision floating point accuracy on state-of-the-art DNNs (i.e. ResNet) and large-scale datasets.
- N-ary formats can potentially contain a large amount of sparsity and, in combination with fixed-point activations, outperform the accuracy of uniformly quantized models (i.e. binarization).

- RaS compression and algorithm is highly efficient in terms of storage as well as inference and even outperforms well-engineered BLAS libraries (i.e. (binary) Eigen, Gemmlowp) with just a few lines of code.

RaS is a promising technique for a wide range of processors, especially since it does not require specialized low-precision formats. While RaS maps well to vectorized and parallel CPUs, the main drawback is load imbalance when mapped to massively-parallel architectures.

Chapter 6

Parameterized Structured Pruning

The previous chapter introduced a novel compression technique in combination with the RaS algorithm, that builds an efficient inference framework for a large variety of general-purpose hardware platforms. While the RaS algorithm is able to elegantly map sparse and non-uniformly quantized neural networks onto parallel-vector units (such as CPUs), it produces load imbalance through fine-grained sparsity that prevents efficient deployment onto massively-parallel processors. These domain-specific accelerators, however, are increasingly used in embedded systems because they significantly outperform their general-purpose counterpart: for instance, NVIDIA's Jetson Nano system is about 26 times faster than ARM's Cortex-A53 on the ResNet-50 model at a similar energy consumption. Consequently, there is a need to implement compression techniques that map well onto these architectures.

Structured pruning can prevent the drawback of load imbalance by inducing sparsity in a hardware-friendly manner, however, pruning structures is not as trivial as individual weights and potentially causes high accuracy degradation. Structured pruning techniques for neural networks from related literature are usually either retraining or regularization based. Retraining techniques measure the error in loss functions between unpruned and pruned weights or activations and aim to minimize the respective errors. Regularization techniques add an l_1 penalty term to the loss function of randomly initialized neural networks, which pushes unimportant structures to zero.

This chapter introduces *Parameterized Structured Pruning (PSP)*, a novel

compression technique based on learnable structure parameters and low-overhead magnitude pruning. Together with regularization, PSP creates a highly flexible framework for massively-parallel processors (such as GPUs and TPUs). This work has been published at the *6th International Conference on Machine Learning, Optimization, and Data Science (LOD)* [6].

6.1 Parameterization

PSP is inspired by magnitude-based pruning methods where weights below a certain threshold are set to zero. This rather simple technique achieves high sparsity rates while being computational very efficient and, hence, can be used dynamically during training, which optimizes the learning process since the weights are not statically set to zero. However, magnitude-based pruning can only be applied to single weights and is not applicable to a whole set of weights within a defined structure. This section introduces a parameterization that ultimately enables magnitude-based pruning on arbitrary defined structures within a neural network.

Let each layer be a abstract processing units, where each unit computes an activation function of the form

$$z = g(\mathbf{W} \oplus \mathbf{x}), \tag{6.1}$$

where $\mathbf{W}$ is a weight tensor, $\mathbf{x}$ is an activation tensor, $\oplus$ denotes a linear operation (e.g. a convolution) and $g(\cdot)$ is a non-linear function. The goal is to learn a structured sparse substitute $\mathbf{Q}$ for the weight tensor $\mathbf{W}$, so that all weights in $\mathbf{Q}$ can be pruned simultaneously. It is vital for the prediction performance to identify the importance of sparse substitutes $\mathbf{Q}$ that contribute less to the objective function and can consequently be pruned. This importance can be learned by parameterizing the substitutes $\mathbf{Q}$ and optimizing them together with the weights of $\mathbf{W}$ using backpropagation. It is therefore necessary to divide the tensor $\mathbf{W}$ into subtensors $\{\mathbf{w}_i\}$ so that each $\mathbf{w}_i = (w_{i,j})_{j=1}^{m}$ contains the m weights of structure i. The subtensor $\{\mathbf{w}_i\}$ is substituted by structured sparse tensor $\mathbf{q}_i$ during forward propagation as

$$\mathbf{q}_i = \mathbf{w}_i \alpha_i \tag{6.2}$$

where α_i is the respective structure parameter of the pre-defined structure i. The gradient of α_i is calculated, following the chain rule as

$$\frac{\partial E}{\partial \alpha_i} = \sum_{j=1}^{m} \frac{\partial E}{\partial w_{i,j}} \ , \tag{6.3}$$

where E represents the objective function. As a result, the structure parameters α_i descend towards the predominant direction of the weights within structure i. Furthermore, structures that contribute more to the objective function are represented with high magnitudes while low-contributing structures are represented with low magnitudes. Consequently, all α_i are optimized together with the weights in their respective structure i but can be regularized and pruned independently. It should be noted that the introduced auxiliary parameters require additional memory and computation, however, they are folded into the weight tensor $\mathbf{W}$ before inference, resulting in no memory or compute overhead during deployment.

6.2 Regularization

Training over-parameterized neural networks without regularization produces large weights which result in an unstable network, since small variations in the input data can result into large changes in the output predictions. Such a behaviour leads to over-fitting models because they memorizes these small variations in training samples and, ultimately, performs well on the learned data but poorly on unseen samples. Furthermore, large-magnitude weights have a lower effective learning rate that additionally reduces the generalization performance.

As a consequence, the learning algorithm needs to be modified by changing the loss function, in order to encourage the neural network to use smaller weights. This is primarily done by adding an term to the loss function that penalizes parameters. Here, the most popular choices are the ℓ_1 and ℓ_2 vector norm that are similar but produce significantly different models. Both norms limit the growth of the weights and, consequently, reduce the complexity of the neural network, which can be also leveraged for pruning the structure parameters α_i. The ℓ_1 norm is based on the absolute magnitude of the structure parameters and

is added to the loss function as

$$E_{\ell_1}(\alpha_i) = E(\alpha_i) + \lambda|\alpha_i|, \tag{6.4}$$

where λ is the regularization strength. This modification of the loss function - for SGD based learning algorithms - changes the update rule for the structure parameters to

$$\Delta\alpha_i(t+1) = -\eta\frac{\partial E}{\partial \alpha_i} - \lambda\eta\,\mathrm{sign}(\alpha_i), \tag{6.5}$$

where η is the learning rate. The ℓ_2 norm is based on the squared magnitude of the structure parameters, where the loss is calculated as

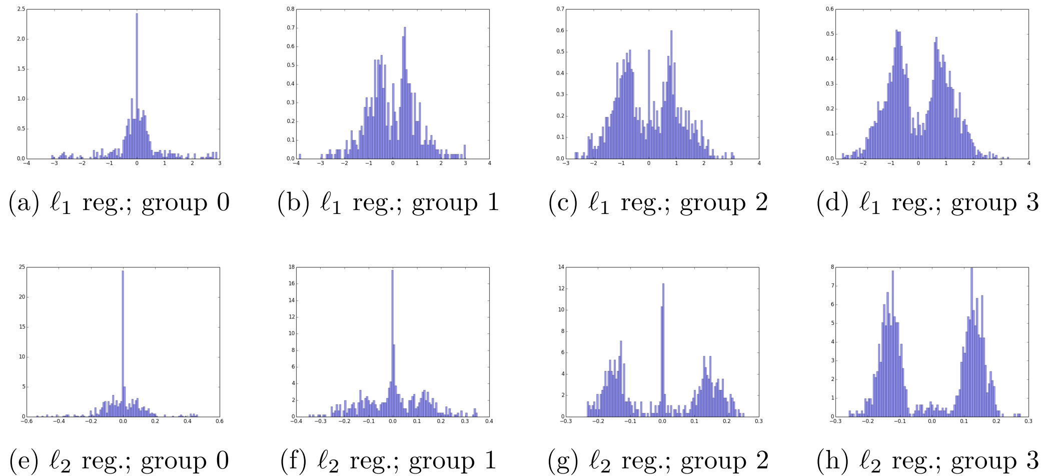
$$E_{\ell_2}(\alpha_i) = E(\alpha_i) + \frac{\lambda}{2}\alpha_i^2. \tag{6.6}$$

Based on the derivative of the ℓ_2 norm, the gradient updates are calculated as

$$\Delta\alpha_i(t+1) = -\eta\frac{\partial E}{\partial \alpha} - \lambda\eta\alpha_i. \tag{6.7}$$

The gradient for the ℓ_1 term is undefined at zero and one everywhere else. As a consequence, ℓ_1 regularization only considers the direction of the parameters while ℓ_2 regularization also takes the magnitude into account. This makes the ℓ_2 term to the defacto standard regularizer and it is used in all popular neural network architectures as well as training techniques.

The proposed parameterization in combination with regularization shrinks the complexity (variance of the defined structures) of a neural network, where the magnitude of the structure parameters indicate its importance. This shrinking of the structures can be best visualized using the parameter values: Figure 6.1a-6.1d shows the ℓ_1 regularized structure distributions. As can be seen, a large amount of parameters are close to or exactly zero, but there is no clear border which separates important and unimportant structures. However, if the parameters are regularized with the ℓ_2 term, the structures form unimodal, bimodal and trimodal distributions (Fig. 6.1e-6.1h), indicating a clear distinction between important and unimportant structures parameters. The ℓ_2 penalty term ultimately seems to be the better regularizer for the required structure separation, but it has the drawback that many structures are pushed only close to zero, while the ℓ_1 penalty term pushes them exactly to zero.



(a) ℓ_1 reg.; group 0 (b) ℓ_1 reg.; group 1 (c) ℓ_1 reg.; group 2 (d) ℓ_1 reg.; group 3

(e) ℓ_2 reg.; group 0 (f) ℓ_2 reg.; group 1 (g) ℓ_2 reg.; group 2 (h) ℓ_2 reg.; group 3

Fig. 6.1 Different distributions of column-wise structure parameters with ℓ_1 and ℓ_2 regularization of a fully trained ResNet with 18 layers on ImageNet. The distributions correspond to the first convolution in the first block in the respective group. **No** pruning was performed ($\epsilon = 0$). Note that peaks visually close to zero are not exactly zero. [6]

6.3 Pruning

While ℓ_1 regularization implicitly sparsifies parameters, ℓ_2 regularized parameters require explicit sparsification that can be performed using threshold-based magnitude pruning. Let ν_i be the regularized *dense structure parameter* associated with structure i, then the *sparse structure parameter* α_i is obtained as:

$$\alpha_i(\nu_i) = \begin{cases} 0 & |\nu_i| < \epsilon \\ \nu_i & |\nu_i| \geq \epsilon \end{cases}, \tag{6.8}$$

where ϵ is a hyperparameter that defines the pruning threshold. This threshold function can be applied after training the neural network in order to obtain a static pruning mask, which can be used to statically set certain structures to zero before fine tuning the sparse network. However, it can not be used in this form dynamically during training, as the function is not differentiable at $\pm\epsilon$ and the gradient is zero in $[-\epsilon, \epsilon]$, making the network harder to train. Similar to quantization functions, the gradient of α_i needs to be approximated by defining an STE as:

$$\frac{\partial E}{\partial \nu_i} = \frac{\partial E}{\partial \alpha_i}. \tag{6.9}$$

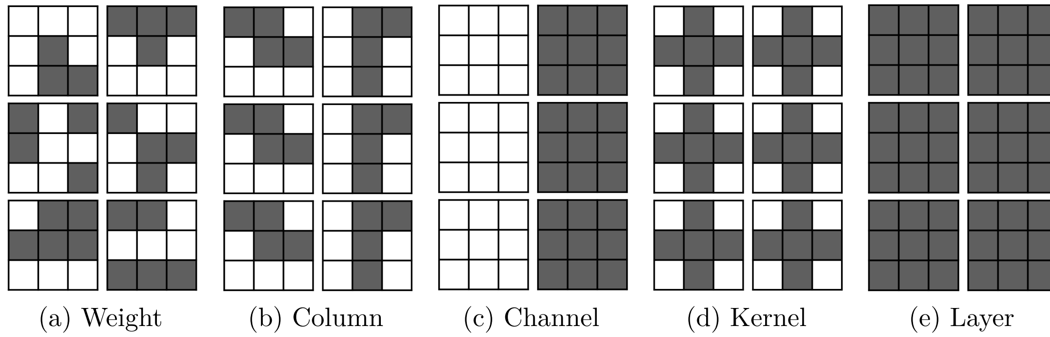


Fig. 6.2 Illustration of fine-grained (Fig. 6.2a) and several structured forms of sparsity (Fig. 6.2b-6.2d) for a 4-dimensional convolution tensor. The large squares represent the kernels, and the corresponding horizontal and vertical dimensions represent the number of input feature and output feature maps, respectively. The computation of all structured forms of sparsity can be lowered to matrix multiplications (independent of stride and padding). [6]

This technique enables the use of the sparse parameter α_i for forward propagation and the dense parameter ν_i for backward propagation. As a consequence, the dense structure parameters ν_i are updated instead of the sparse structure parameters α_i, improving convergence and, ultimately, resulting in a better performance because improperly pruned structures can reappear if ν_i moves out of the pruning interval $[-\epsilon, \epsilon]$.

6.4 Hardware-Friendly Structures

Let the convolution operator be defined with the hyperparameters R, S, C and K, where $R_\times S$ is the filter kernel size, C the number of input and K the number of output feature maps per layer. This section discusses the potential use of these hyperparameters in order to create structure tensors that are, not only beneficial for hardware, but also for creating a large amount of sparsity. The targeted hardware platforms are highly-parallel architectures such as domain-specific accelerators based on systolic arrays, general-purpose GPUs as well as data-flow architectures on FPGAs.

The most fine-grained structure definition is to assign each weight w_i a parameter α_i, where $\alpha \in \mathbb{R}^{R \times S \times C \times K}$. This structure definition is obviously equal to simple weight pruning and produces unstructured sparsity in the weight tensor (see Fig. 6.2a), which has poor hardware properties.

6.4.1 Column Pruning

Convolutions are usually lowered to matrix multiplications in order to leverage data locality and reuse of specialized hardware architectures (systolic arrays) and highly optimized BLAS libraries. Here, the *im2col* technique is applied, where discrete input blocks (depending on filter size and stride) are duplicated and reshaped into columns of two dimensional matrices. These auxiliary matrices are usually significantly larger then the discrete inputs as well as the weight tensors and are, therefore, the most memory consuming part when performing convolutions. The structure tensor can be defined to $\alpha \in \mathbb{R}^{R \times S \times C}$, so that whole columns in the flatted weight tensor are sparsified and the respective row of the input data can be removed. For this individual structure, the corresponding gradient is calculated as:

$$\partial E / \partial \alpha_{r,s,c} = \sum_{k=1}^{K} \partial E / \partial W_{k,c,r,s} \tag{6.10}$$

This structure sparsification is denoted as column pruning (see Fig. 6.2b) and has many desirable properties: (i) The structure granularity is relatively fine, promising a large amount of sparsity. (ii) The resulting operation is a dense matrix multiplication, which is inline with highly parallel hardware and BLAS algorithms. (iii) It compresses not only parameters and removes computations, but also compresses the extremely high memory requirements of auxiliary matrices.

6.4.2 Channel Pruning

While column pruning features excellent hardware properties, its applicability to commercial available accelerators (for instance NVidia GPUs or TPUs) is challenging because they offer an opaque software stack, that allows user to only implement convolution operators as black boxes. It is possible to write and optimize CUDA routines for the required operators but highest performance depends on the usage of the cuDNN and cuBLAS library. On the other hand, one could argue that *im2col* can be implemented in a CUDA routine, followed by subsequently calling the matrix multiplication operator of cuBLAS. This, however, would result in inferior performance, since cuDNN internally implements *im2col* into the shared memory of the streaming multiprocessors which reduces global memory requirements and bandwidth dramatically. In order to leverage these

processors and their libraries, it is necessary to prune all weights connected to an input or output feature map, so that the whole channels and their corresponding feature maps can effectively be removed (see Fig. 6.2c). The structure for input and output channel are defined by $\alpha_{in} \in \mathbb{R}^C$ and $\alpha_{out} \in \mathbb{R}^K$, where the respective gradients are calculated as:

$$\partial E/\partial \alpha_c = \sum_{k=1}^{K} \sum_{r=1}^{R} \sum_{s=1}^{S} \partial E/\partial W_{k,c,r,s},$$
$$\partial E/\partial \alpha_k = \sum_{c=1}^{C} \sum_{r=1}^{R} \sum_{s=1}^{S} \partial E/\partial W_{k,c,r,s}.$$

Channel pruning is not only beneficial for the software stack but also for reducing the memory requirements of discrete inputs to the convolution operator, since feature maps can be removed completely. Very deep neural networks implement various branching techniques in order to avoid the vanishing gradient problem that comes along with deepening the networks. These branches may constrain or even prevent channel pruning at certain positions, because feature maps are connected to multiple convolutional layers. Resulting implications and a novel solution for this issue is introduced and discussed in detail in Chapter 7.

6.4.3 Kernel Pruning

The typical kernel shapes of convolution operations in modern neural architectures are 3×3, 5×5 or 7×7, depending on the estimated or evaluated distance of correlations in the feature maps. These dense and quadratic kernels consume a large amount of computations and parameters, although non-quadratic and sparse kernels (see Figure 6.2d) could potentially perform similar well but with much fewer resources, which can also be observed with dilated convolution. In order to create such sparse kernels with PSP, the structure tensor is defined by $\alpha \in \mathbb{R}^{R \times S}$, where the gradient is calculated as:

$$\partial E/\partial \alpha_{r,s} = \sum_{k=1}^{K} \sum_{c=1}^{C} \partial E/\partial W_{k,c,r,s}. \tag{6.11}$$

Sparse kernels allow larger receptive fields with fewer layers while - depending on the compression rate - not requiring more parameters and computations. This form of sparsity is especially promising in data-flow architecture, where direct

convolutions can be implemented with dedicated hardware configurations for each layer.

6.4.4 Layer Pruning

DNNs require different stages, where the effective receptive field is varied in order to capture various correlations in the features. Instead of increasing the filter kernels, which would quickly violate the given resources of hardware platforms, the feature resolution is changed through pooling and striding. The standard setting for large-scale image classification using ImageNet is to half the feature resolution in each stage from 224×224, to 56×56, to 28×28, to 14×14 and to 7×7, before the resulting features are forwarded to the global-average layer. The amount of blocks (or layers) per stage is usually not constant: for instance, the popular ResNet-50 architecture uses $1-3-4-6-3$ blocks in their respective stages. Finding such configurations in an iterative manner is extremely time consuming and requires expert knowledge on the respective data. An alternative technique is to simply initialize all stages with the same amount of layers and finally prune the relatively unimportant layers. This can be done by defining the structure tensor as a scalar ($\alpha \in \mathbb{R}$) and calculating the gradient as:

$$\partial E/\partial \alpha = \sum_{k=1}^{K} \sum_{c=1}^{C} \sum_{r=1}^{R} \sum_{s=1}^{S} \partial E/\partial W_{k,c,r,s}, \tag{6.12}$$

which consequently prunes the layers with the lowest gradients. Pruning whole layers is obviously beneficial for every hardware platform and can be used together with other sparse structures (i.e. kernels or channels).

6.5 Experiments

6.5.1 Evaluating PSP

This section evaluates the effectiveness of PSP and validates the proposed techniques through extensive comparison to other schemes. The evaluation is performed on a ResNet model with 56 layers using the CIFAR-10 dataset and columns (Fig. 6.2b) are set as the targeted structured sparsity. Fig. 6.3 reveals the overall experiment using several decoupled elements of PSP. The moving

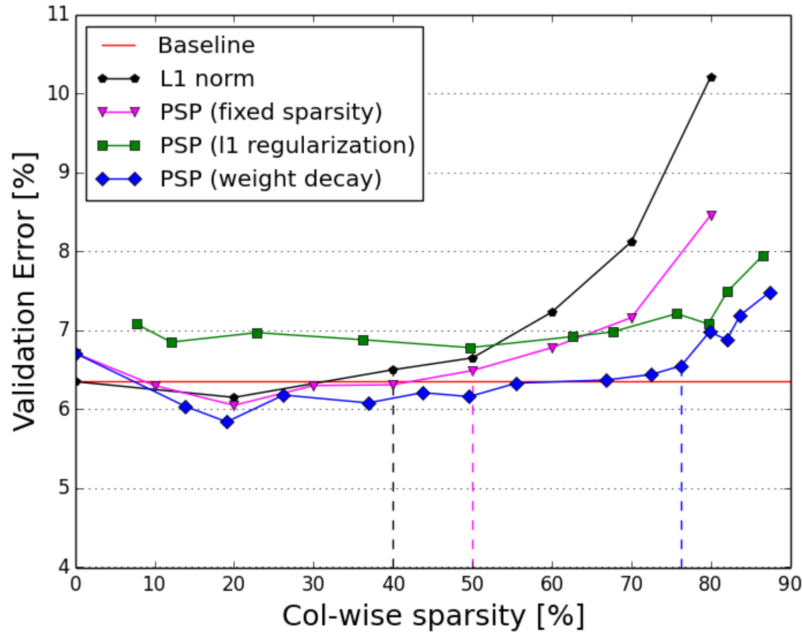


Fig. 6.3 ResNet network with 56 layers on CIFAR10 and column pruning. [6]

average of the test error is reported using a single run, in order to save training time of the models. Baseline accuracy of the ResNet-56 is 6.35% and is shown as red line in the plot. The dashed vertical lines indicate the maximum amount of sparsity while the baseline accuracy (±0.25%) is still maintained.

A popular metric for evaluating the importance of structures is the l_1 norm, which is calculated as:

$$||w_i||_1 = \sum_{j=1}^{m} |w_{i,j}|, \tag{6.13}$$

for the m weights within structure i. Structures can be simply compared using the magnitude of this norm and, following a threshold heuristic, subsequently pruned. The first experiment analyses the effectiveness of the proposed parameterization of structures compared to a standard structure evaluation using the l_1 norm. Both techniques are implemented using the same threshold heuristic in order to give an fair comparison and PSP is used without regularization ($\lambda = 0$). Here, the thresholds are chosen dynamically during training in a way that each layer obtains a pre-defined amount of sparsity. The sparsity level per layer is varied in $[0\%, 90\%]$ with 10% step size. As can be seen, the importance evaluation of structures through parameterization clearly outperforms the l_1 norm technique. This performance benefit can be explained by the amount of

threshold changes during optimization: the parameterization technique only updates a single parameter per iteration that can change weather a structure is pruned or not. On the contrary, there are m possible parameter updates per iteration that can influence this for the l_1 norm technique. Especially for dynamic pruning processes, a stable structures selection is necessary in order to guarantee convergence.

The previous experiment uniformly distributed the sparsity of the model by fixing the sparsity in each layer to the same amount, which is not optimal since sparsity is naturally distributed heterogeneously. In this experiment, the threshold function of Equation 6.8 in combination with a static threshold parameter ϵ is used in order to allow such heterogeneous sparsity distributions. Without explicit regularization this would result in very low pruning rations, since the structure parameters α would simply move out of the pruning interval $[-\epsilon, \epsilon]$. Thus, $l1$ and $l2$ penalties are applied on α in order to push them towards zero. Since these regularization terms feature different pruning properties (see Section 6.2), the experimental setup needs to be carefully selected accordingly: for the $l1$ penalty, the threshold is fixed to $\epsilon = 10^{-3}$, with an initial regularization strength of $\lambda = 10^{-10}$ that is increased by a factor of 10 for each sparsity level. For the $l2$ penalty, the regularization strength is fixed to $\lambda = 10^{-4}$, with an initial threshold $\epsilon = 0.0$ that is increased by $2 \cdot 10^{-2}$ for each sparsity level. The comparison between $l1$ and $l2$ regularization reveals, that $l2$ consistently outperforms $l1$. Even for very small pruning ratios, the $l1$ techniques performs worse than the baseline model, which can be explained by the findings in Section 6.2. More specifically, the gap to the baseline accuracy is caused by a lower effective learning rate (compared to $l2$) for the model, that consequently reduces the generalization performance.

6.5.2 Pruning different structures

Section 6.4 discusses several hardware-friendly structures with varying granularity and targeting various massivel- parallel processor types, such as domain-specific accelerators based on systolic array, general-purpose GPUs and data-flow architectures on reconfigurable logic. This section evaluates the introduced structure granularities from a theoretical perspective and analyzes the potential through comparing the amount of layers, parameters as well as MAC operations. The

Table 6.1 Column-, channel-, shape- and layer-pruning using PSP, validated on DenseNet40 ($k = 12$) on the CIFAR10 dataset. M and G represents 10^6 and 10^9, respectively. [6]

Model	Layers	Parameters	MACs	Error [%]
Baseline	40	1.02M	0.53G	5.80
Column pruning	40	0.22M	0.10G	5.76
Channel pruning	40	0.35M	0.18G	5.61
Kernel pruning	40	0.92M	0.47G	5.40
Layer pruning	28	0.55M	0.28G	6.46
Layer+channel pruning	33	0.48M	0.24G	6.39

evaluation is performed on the example of a DenseNet architecture, with 40 layers and a growth rate of $k = 12$ on the CIFAR-10 dataset. The regularization strength is set to $\lambda = 10^{-4}$ and the pruning threshold to $\epsilon = 0.1$. Table 6.1 reveals the results of the structure evaluation.

As can be seen, the best performing structure is column pruning, which is able to compress the model by roughly a factor of 5 while not degrading the prediction performance. The high compression rate is very promising since the structure is inline with the demand of recent trends towards systolic-array based accelerators.

Channel pruning performs slightly worse, but is still able to compress the model by a factor of 3 with a small improvement in test accuracy. This compression rate reflects the possible performance improvements on general-purpose GPUs, where sparse columns can not be leveraged efficiently.

Kernel pruning seems to have rather bad compression properties, which is surprising considering the popularity of dilated convolutions that feature sparse kernels explicitly. However, this structures obtains the best test accuracy, indicating that the hyperparameters ϵ and λ are not optimal, but fine tuning these parameters for each structure requires too much training time for this kind of evaluation.

Layer pruning is able to remove 12 layers from the model, but at the same time, shows the worst accuracy degradation of all structure, which is not surprising due to the extreme coarse structure. On the other hand, when combined with channel pruning, it reduces less layers while compression the overall model more and achieving better accuracy. This might be a good solution for processor types

that require large amounts of parallel computations.

6.5.3 CIFAR-10/100 and ImageNet

Section 6.5.2 evaluated possible hardware-efficient structures for pruning, where column sparsity clearly achieves the highest compression rates while excellently mapping onto systolic-array based accelerators. This section leverages column sparsity on different neural architectures with varying configurations and several datastets, in order to evaluate the performance of PSP while simultaneous considering data complexity, neural architecture and over parameterization. The regularization strength is fixed to $\lambda = 10^{-4}$ and the pruning threshold to $\epsilon = 0.1$ for DenseNet and $\epsilon = 0.2$ for ResNet architectures.

Neural Architectures: Section 2 introduced the evolution of neural designs, where two groundbreaking techniques are discussed that prevent the vanishing gradient problem: residual and dense connectivity between layers. This experiment aims to measure the resource efficiency of both techniques in terms of required MACs and parameters. Structured pruning is the perfect tool for such experiments, since it compresses the standard architectures to a minimum while maintaining the structure for efficient hardware deployment. The results are summarized in Table 6.2 for CIFAR-10/100 and 6.3 for ImageNet. As can be seen, dense connectivity outperforms residual connections for all datasets. There is especially one interesting tendency to observe: DenseNet is increasingly superior to ResNet with increasing complexity of the dataset. While both techniques successfully prevent vanishing gradients during optimization, dense connections allow for higher feature reuse, which ultimately reduces the amount of required computations and parameters. It should be noted, that dense connections can in practice potentially be inefficient due to load imbalance and high activation requirements. These drawbacks are further discussed in detail in Chapter 7.

Another popular technique for designing neural architectures are the bottleneck layers, where 1×1 convolutions are applied to reduce the amount of features before entering layers with larger kernels. This reduces the amount of required computations and parameters but increases the number of layers and activations. Structured pruning can obliterate the need for these bottleneck layers, since computations and parameters can be reduced explicitly. For instance, the DenseNet121 network in combination with PSP and without bottleneck

Table 6.2 ResNet and DenseNet on CIFAR10/100 using column pruning. M and G represents 10^6 and 10^9, respectively. [6]

Model	Layer	Parameter	MACs	Error [%]
CIFAR10				
ResNet	56	0.85M	0.13G	6.35
ResNet-PSP	56	0.21M	0.03G	6.55
DenseNet ($k = 12$)	40	1.02M	0.27G	5.80
DenseNet-PSP ($k = 12$)	40	0.22M	0.05G	5.76
DenseNet ($k = 12$)	100	6.98M	1.77G	4.67
DenseNet-PSP ($k = 12$)	100	0.99M	0.22G	4.87
CIFAR100				
ResNet	56	0.86M	0.13G	27.79
ResNet-PSP	56	0.45M	0.07G	27.15
DenseNet ($k = 12$)	40	1.06M	0.27G	26.43
DenseNet-PSP ($k = 12$)	40	0.37M	0.08G	26.30
DenseNet ($k = 12$)	100	7.09M	1.77G	22.83
DenseNet-PSP ($k = 12$)	100	1.17M	0.24G	23.42

removes 2.6× parameters, 4.9× MACs and 1.9× layers while only degrading top-5 accuracy by 2.3%.

Over parameterization: in order to increase the prediction performance of neural networks, the architectures need to be enlarged by either widening or deepening the respective model. This enlargement usually tremendously increases the required computation and memory requirements of the model, due to heavy over parameterization. In this experiment, the DenseNet architecture is increased from 40 to 100 layers, which reduces the test error by 1 % at the cost of 7× more computations and parameters. After pruning the models, the cost for the 1% performance improvement is only 4× more computations and parameters. This highlights the ability of PSP to adopt automatically to the over parameterization without any hyperparameter tuning.

Dataset complexity: similar to over parameterization is the increase of data complexity, where either the input resolution is changed, the amount of training samples or prediction classes. For instance, the CIFAR-10 and CIFAR-100 datasets have similar samples but they feature 10 and 100 classes, respectively, which changes the prediction complexity. When trained using the same model (e.g. ResNet-56) and pruned, it can be observed that PSP automatically adapts to

Table 6.3 ResNet and DenseNet on ImageNet using column pruning. [6]

Model	Layer	Parameters	MACs	Top-1 [%]	Top-5 [%]
ResNet-B	18	11.85M	1.82G	29.60	10.52
ResNet-B-PSP	18	5.65M	0.82G	30.37	11.10
ResNet-B	50	25.61M	4.09G	23.68	6.85
ResNet-B-PSP	50	15.08M	2.26G	24.07	6.69
DenseNet-BC	121	7.91M	2.84G	25.65	8.34
DenseNet-BC-PSP	121	4.38M	1.38G	25.95	8.29
DenseNet-C	63	10.80M	3.05G	28.87	10.02
DenseNet-C-PSP	63	3.03M	0.58G	29.66	10.62
DenseNet-C	87	23.66M	5.23G	26.31	8.55
DenseNet-C-PSP	87	4.87M	0.82G	27.46	9.15

the complexity of the task. As a result, the CIFAR-10 model is $\sim 2\times$ smaller than the CIFAR-100 model, while both compressed models maintain the prediction performance of the baseline model.

6.6 Summary

In order to effectively compress DNNs, quantization and pruning are usually considered. However, unconstrained sparsity usually leads to unstructured parallelism, which maps poorly to massively-parallel processors and substantially reduces the efficiency of general-purpose processors. Similar applies to quantization which often requires dedicated hardware. This section introduced PSP, a novel structured pruning technique which reduces memory and compute requirements while creating a form of sparsity that is inline with massively-parallel processors. PSP exhibits parameterization of arbitrary structures in a weight tensor and uses weight decay to force certain structures towards zero, while clearly discriminating between important and unimportant structures. Combined with threshold-based magnitude pruning and backward approximation, PSP can remove a large amount of structure while maintaining prediction performance. The key insights of this section are:

- Processors and software stacks have different requirements in terms of structure within computations and do not benefit from unstructured sparsity.

- Structured pruning is able to generate computations that are inline with such accelerators and the proposed PSP technique features large degrees of freedom when defining structures.
- The amount of sparsity within a model depends on the over parameterization of the model, neural architecture as well as complexity of the task.

PSP is a highly flexible tool for compressing DNNs while targeting a certain processor architecture, especially because it can be combined with quantization. Using this techniques is ultimately a trade-off between training and inference time, since optimal submodels can be found in the larger baseline architecture.

Chapter 7

Architecture Search

PSP greatly removes unimportant structures dynamically during training and it is a highly flexible framework that is able to target virtually any digital computing platform. The previous chapter evaluated PSP on ResNet and DenseNet architectures, which have been proposed in related literature and are very popular due to their efficiency and scalability. However, the connectivity in these architectures severely influences channel pruning because features connected to a certain pruned channel can not be simply removed as they may be connected to a channel in another layer. This has two major drawbacks: first, activation memory is not reduced if a feature map is connected to multiple channels. Second, software stacks for most accelerators are not able to leverage such sparse-channel patterns efficiently. Another issues regarding the previously discussed structures is that they are not inline with most libraries for DNNs and, hence, do not map well onto their respective accelerators.

This chapter uses PSP together with architecture search in order to explore the design space and efficiency of neural networks for a specific hardware and software architecture. NVIDIA's Xavier and Nano processors are used as evaluation platforms using the TensorRT compiler framework in order to achieve high performance and productivity on these devices. The structures for PSP are adjusted to the demands of the cuDNN library and analyzed using several performance critical metrics as well as benchmarks. Last, the sigmoidal building block is introduced, a novel connectivity for DNNs that is motivated by residual connections, but allows for more efficient channel pruning.

The methodology introduced in this chapter does not only results into efficient inference models through pruning, but also enables a detailed analyzes of design

principles and techniques for DNNs. The information found using this setup gives valuable insights into the limitation of current architecture principles and shows the potential for future neural designs. More importantly, it shows the impact of activation memory on latency and throughput, which appears to be of higher relevance than parameters and computations.

7.1 Design Space Exploration

Scaling neural networks in width or depth is required to increase their prediction performance. The initial deep architectures are scaled in width rather than depth, which increases the amount of parameters and computations dramatically, since they usually increase quadratic with the width. Inception blocks reduce this quadratically scaling by placing several independent layers in parallel. Scaling the depth rather than the width is promising because computations and memory increase linearly with the depth, however, the backpropagation algorithm quickly becomes a bottleneck as gradients are vanishing for very deep architectures. There have been two major breakthroughs in the architecture design of neural networks that prevent the vanishing gradient problem, which occurs when many layers are stacked sequentially. These breakthroughs are residual and dense connections between layers, enabling much deeper architectures and, therefore, more accurate and efficient models. Virtually any advanced architecture nowadays - found by human or machine design - implements either or both forms of connectivity. While both connectivity patterns target the same issue, there impact on efficiency as well as design space opportunities differ significantly. This section details about the implications of these building blocks and explores possible design space decisions.

7.1.1 Building Blocks

Machine learning frameworks and compilers - including highly optimized libraries such as TensorFlow and TensorRT - constrain the user to a certain degree of freedom in terms of designing operators. This section explores the most popular building blocks for DNNs and explains their possible design configurations within these frameworks. The design parameters include filter width and height ($k \times k$),

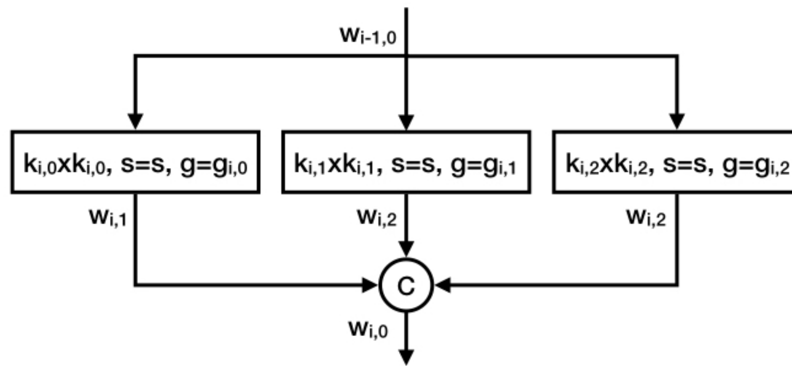


Fig. 7.1 Inception block.

stride (s), group size (g), number of features w as well as element wise tensor addition, split and concatenation operations.

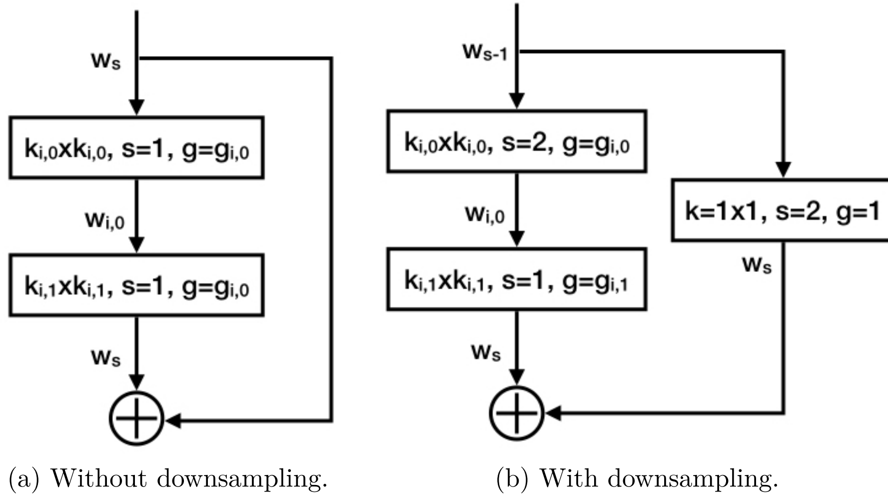
Inception

Figure 7.1 illustrates the inception block, which is based on several parallel and independent layers with varying filter sizes (usually 1×1, 3×3 and 5×5 kernels), operating on the same features. The resulting feature maps are concatenated into a large activation tensor which is forwarded towards the next block. The primary aim of the inception block is to capture multiple correlations in the feature space while keeping the network depth relatively shallow, in order to avoid vanishing gradients. Of course, the same could be achieved with a single - large enough - layer as well, however, the inception block is significantly smaller in terms of parameters and computations.

Inception blocks offer a large amount of possible configurations: each block i allows arbitrary filter kernels $k_{i,0} \times k_{i,0}$, $k_{i,1} \times k_{i,1}$, $k_{i,2} \times k_{i,2}$, any group size in $1 \leq g_i \leq w_{i-1,0}$, as well as any number of output feature maps for each convolution $w_{i,1}$, $w_{i,2}$, $w_{i,3} \geq 0$. An appropriate solution for these configurations can be found explicitly by using group, kernel and channel pruning. The number of output feature maps for the whole block is implicitly defined by the parallel convolutions $w_{i,0} = w_{i,1} + w_{i,2} + w_{i,3}$.

Residual Connections

While Inception blocks enable a resource indulgent width scaling of the architecture, they lack the ability of depth scaling due to vanishing gradients. However,



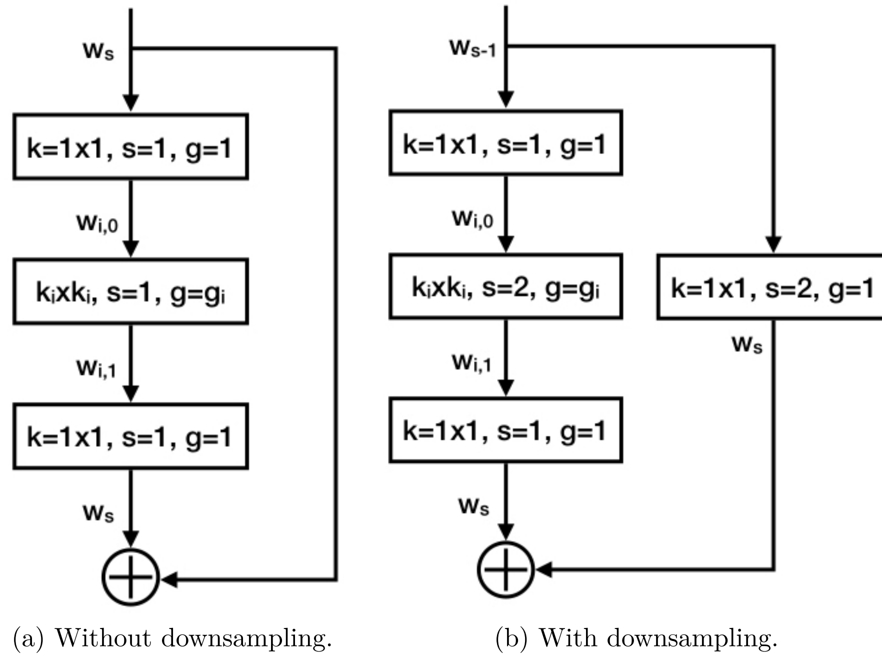
(a) Without downsampling. (b) With downsampling.

Fig. 7.2 Basic residual block.

depth scaling is usually much more beneficial in terms of computations, parameters and accuracy. Residual blocks enable extreme depth scaling of architectures through connections between layers, which ultimately prevent the gradients from vanishing. Figure 7.2 illustrates the basic version of residual blocks - composed of two convolution layers - computing residuals that are element wise added to the shortcut connection.

The design space of the basic blocks is relatively restricted in comparison to Inception blocks, because of the introduced residual connections. These enforce equal sizing of input and output features to the block within a stage to w_s, which is defined by the output of the 1×1 convolution, responsible for down sampling the features. Furthermore, w_s needs to be set statically and can not be found dynamically through pruning, because the additive term prevents feature maps from being set to zero throughout the stage. This means that only $w_{i,0}$ can be found through pruning and all w_s need to be tuned by human or algorithmic design (i.e. reinforcement learning). Group sizes $g_{i,0}$, $g_{i,1}$ and kernel sizes $k_{i,0} \times k_{i,0}$, $k_{i,1} \times k_{i,1}$ are independent of the residual connections and can be pruned as well.

Another popular version is the standard residual block (see Figure 7.3), where 1×1 convolutions are placed before and after the $k_i \times k_i$ layer. In comparison to the basic block, the standard block reduces parameters and computations while it enables a large amount of features in the model, consequently increasing activation requirements.



(a) Without downsampling. (b) With downsampling.

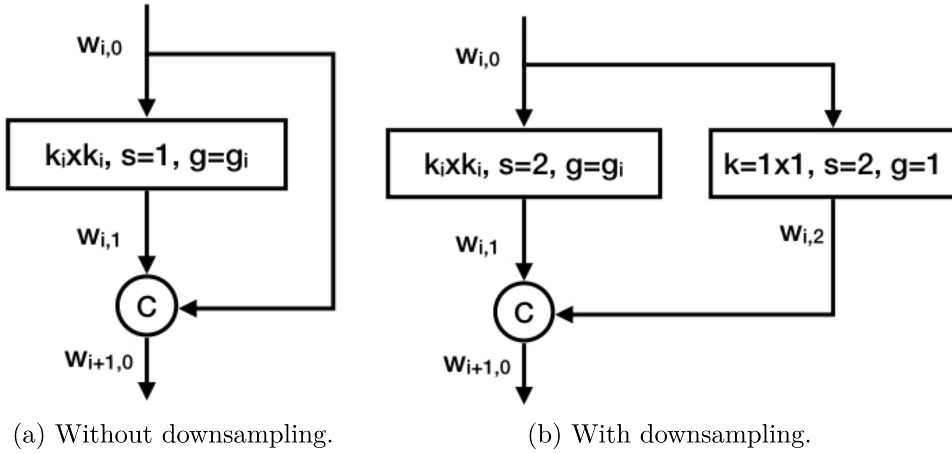
Fig. 7.3 Standard residual block.

For the standard block $w_{i,0}$, $w_{i,1}$ as well as g_i and $k_i \times k_i$ can be learned through pruning, while w_s needs to be set statically. The 1×1 convolutions are usually not combined with grouping, since these layers are very parameter and computation efficient.

Dense Connections

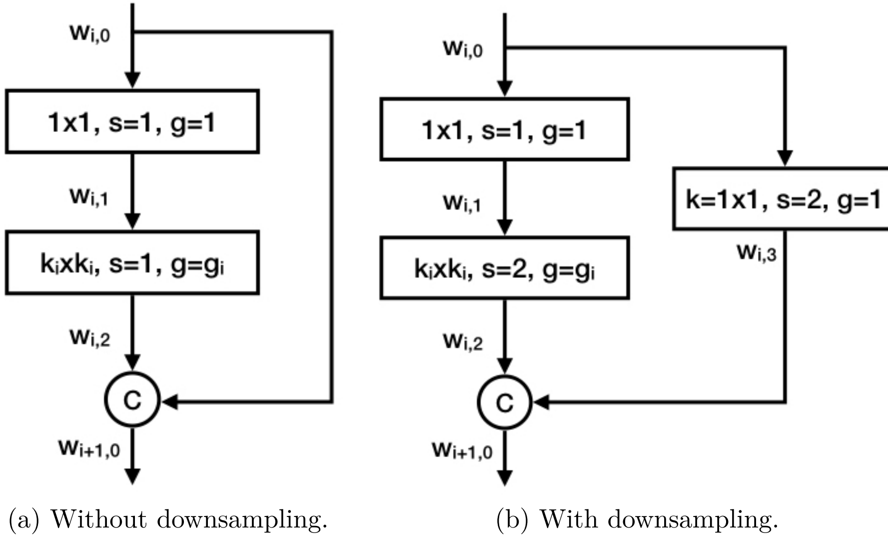
Motivated by residual blocks, dense architectures connect each layer to every other layer in the model, enabling excellent gradient flow during training. As illustrated in Figure 7.4, the basic dense block concatenates input to output features of each layer, enforcing feature sharing within the model, which positively affects parameter and computation efficiency. However, this requires a large amount of activation memory, since all features within the model are forwarded incrementally to the subsequent layers.

The dense connectivity offers a larger design space than the relatively restrictive residual connections through concatenating instead of element-wise adding the activation tensors. Here, the number of features $w_{i,1}$, group size g_i and kernel size $k_i \times k_i$ can be learned through structured pruning. The number of input features to the block is implicitly specified by the previous block



(a) Without downsampling. (b) With downsampling.

Fig. 7.4 Basic dense block.



(a) Without downsampling. (b) With downsampling.

Fig. 7.5 Standard dense block.

$w_{i,0} = w_{i-1,1} + w_{i-1,0}$ or $w_{i,0} = w_{i-1,1} + w_{i-1,2}$.

Because of the linear scaling of input features maps to the block $w_{i,0}$, the dense architecture becomes extremely resource-demanding for very deep models. The standard dense block aims to resolve this issue by reducing the number of input feature through a 1×1 convolution before the expensive $k_i \times k_i$ layer to a constant factor $w_{i,1}$. This works well for parameters and computations but does not reduce the activation number.

The standard block achieves the same design space as the basic block: all block parameters $w_{i,1}$, $w_{i,2}$, $w_{i,3}$ and g_i, $k_i \times k_i$ can be learned explicitly while $w_{i,0}$ results implicitly from the previous block.

7.2 Evaluating the Efficiency of Building Blocks through Camuy

The design space exploration from previous section shows a large variety of possible solution for creating neural architectures and compressing them through structured pruning. While Inception and Dense building blocks allow a large design space and excellent pruning opportunities, Residual blocks constrain the degrees of freedom and require additional static adjustments. This section evaluates these building blocks on massively-parallel processors using the Camuy emulation framework, in order to explore their implications when mapped onto hardware. The content of this section has been published in collaboration with Kevin Stehle at the *Workshop on IoT, Edge, and Mobile for Embedded Machine Learning (ITEM), collocated with ECML-PKDD* [7].

7.2.1 Camuy

Camuy is a lightweight model of a weight-stationary systolic array for linear algebra operations that is integration into the machine learning tool TensorFlow through custom operators. It allows quick explorations of different neural architectures and configurations (such as systolic array dimensions) and estimates required cycles, data movement costs, as well as array utilization. The tool can assist hardware experts to design processors arrays based on a certain neural designs, and machine learning experts to design neural architectures that fit well onto a certain processor.

Camuy is designed following a weight-stationary systolic array, which is a promising candidate to address the increasing costs of data movements and implements a large amount of parallel Processing Elements (PEs). In more detail, the basic architecture of the emulation framework is modelled very similar to Google's TPUv1. Here, the activations flow horizontally and partial sums vertically through the PE array. Each PE is modelled with four data registers: two weight registers to support double buffering, one activation register, and output register for the partial sum. Activations are forwarded from memory through FIFOs to the PEs. Computations for the emulation are performed multi-threaded and vectorized on CPU using the Eigen library.

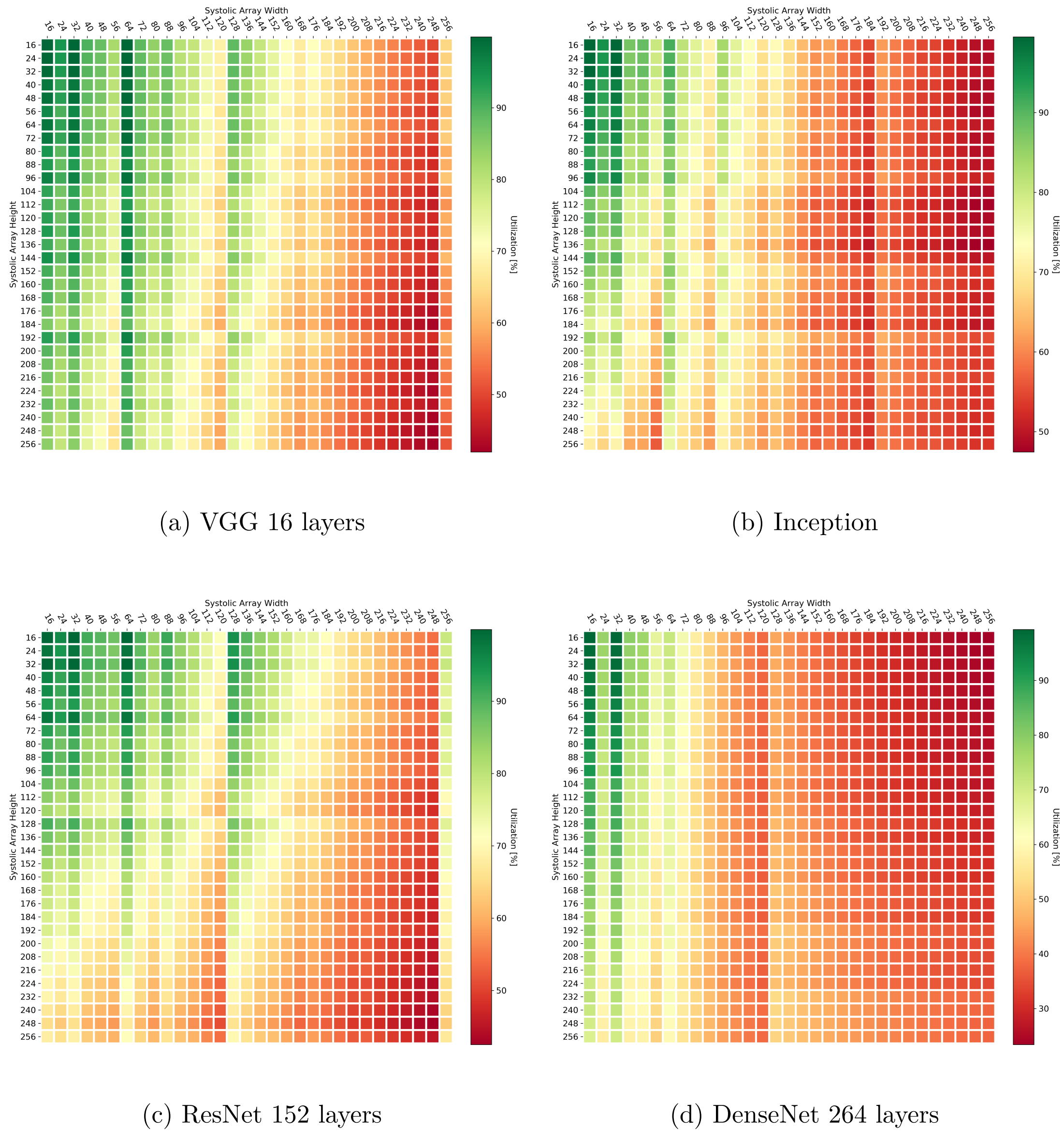
7.2.2 Evaluation

Four architectures are exemplary chosen for the evaluation where each model covers a different design solution: VGG16 and Inception represent rather shallow models that are scaled in width, while ResNet-152 and DenseNet-264 represent deep models with their respective form of connectivity. The main objective here is to evaluate the potential performance of the various techniques on parallel hardware. This is done through estimating utilization of systolic arrays with varying height and width, which is emulated using Camuy. Utilization is a measure that shows how well a certain workload can utilize the available compute resources and, hence, is a key metric for applications on massively-parallel hardware. Figure 7.6 shows the evaluated utilization in form of heatmaps.

As can be seen, the VGG-16 architecture achieves a good utilization for systolic arrays of up to 128 PEs wide. This is not surprising since the model applies only a few wide layers and consequently exhibits great parallelization potential. The Inception architecture achieves a slightly worse utilization in comparison to VGG-16, because of varying kernel sizes within each block and a smaller number of features. The ResNet-152 model also shows a good utilization for larger arrays and, therefore, good parallelization potential while also enabling very deep architectures. This can be explained by avoiding the vanishing gradients through element-wise additions of the features, creating homogeneous computations throughout the model. The DenseNet-264 architecture achieves the worst utilization of all models, resulting from high variance of the operands due to many concatenation operations. As a result, ResNet architectures exhibit the best properties for massively-parallel hardware because they enable extreme depth scaling while also featuring homogeneous computations that utilize PEs well.

7.3 Sigmoidal Residuals

The residual block enables very deep architectures, resulting in resource-efficient models, and shows excellent mapping behaviour on parallel processors. However, as discussed in Section 7.1, the building-block design constrains the degrees of freedom for structured pruning. This section introduces *sigmoidal residuals* that are motivated by the original residual technique and able to decrease vanishing



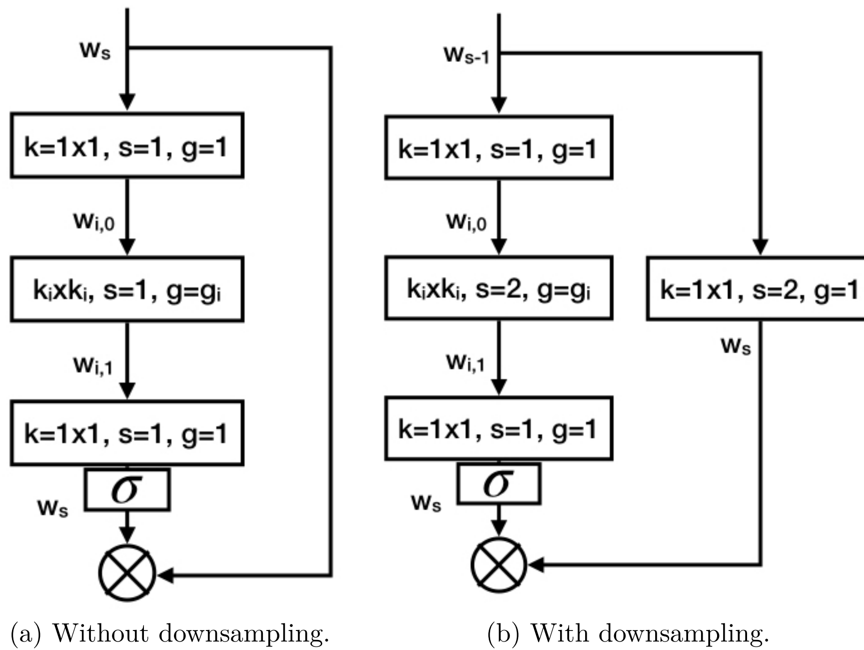
(a) VGG 16 layers

(b) Inception

(c) ResNet 152 layers

(d) DenseNet 264 layers

Fig. 7.6 Systolic array utilization for varying height (y axis) and width (x axis) for several architectures based on heatmaps. [7]



(a) Without downsampling. (b) With downsampling.

Fig. 7.7 Sigmoidal residual block.

gradients, but allow for better pruning opportunities.

7.3.1 Building Block

The original residual building block consists of two or three layers, where the input to the block is connected through a shortcut to the output of a block. This connection is realized using an element-wise addition term on the activations and reduces vanishing gradients. The addition term eliminates pruning opportunities at the interfaces of the block (see w_s in Figure 7.3), because pruned features can be cancelled out by features from consecutive blocks. In order to solve the issue of pruning the interfaces, the sigmoidal residual block (Figure 7.7) replaces the element-wise additive with a multiplicative term. The multiplication consequently enables pruning since all consecutive blocks to a pruned feature will result into a multiplication with a zero-valued feature map.

An important design parameter of such building blocks is the so called *width multiplier*, that states the difference between w_{s-1} and w_s. The most common choice for this parameter is $\frac{w_s}{w_{s-1}} = 2$, resulting in a doubling of features maps for each stage, where all blocks within a stage have the same w_s. Using the sigmoidal block, it is now possible to prune features after the 1×1 convolution layers that are required for increasing the number of feature maps. As a consequence, the

width multiplier $\frac{w_s}{w_{s-1}}$ can be found dynamically during training, resulting in higher compression through a larger design space.

7.3.2 Backward Properties

The sigmoidal block is clearly better suited than the residual block for dynamically learning the architecture through pruning. However, the residual block has certain properties that enable deeper architectures as well as better backward flow. This section explains the differences between sigmoidal and residual block during training and shows their implications with respect to backward properties.

Formally, with $\mathbf{x}_l$ and $\mathbf{x}_{l+1}$ being the input and output vectors of the block l, the residual is defined as:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathbf{W}_l), \tag{7.1}$$

where $\mathcal{F}(\mathbf{x}, \{W_i\})$ denotes the two or three layers within a block to be learned and $\mathbf{x}_l$ the shortcut connection. Stacking several of these blocks in a stage (without a layer for down sampling or width expansion), can be formulated as:

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L} \mathcal{F}(\mathbf{x}_i, \mathbf{W}_i), \tag{7.2}$$

where L states the number of blocks per stage. Based on this formulation and following the chain rule of backpropagation, the backward path can be formulated as:

$$\frac{\partial E}{\partial \mathbf{x}_l} = \frac{\partial E}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial E}{\partial \mathbf{x}_L}(1 + \frac{\partial}{\partial \mathbf{x}_l}\sum_{i=l}^{L} \mathcal{F}(\mathbf{x}_i, \mathbf{W}_i)). \tag{7.3}$$

The gradient $\frac{\partial E}{\partial \mathbf{x}_l}$ can be split into two terms, where the first term $\frac{\partial E}{\partial \mathbf{x}_L}$ is independent of the architecture's depth L. As a consequence, this term is not influenced by the layers within the residual blocks and allows that information is directly propagated to any block within the stage, ultimately enabling the design of very deep architectures.

The backward properties of the sigmoidal block differ to the residual block, due to the change from the additive to the multiplicative term. Formally, the sigmoidal building block calculates the output vector $\mathbf{x}_{l+1}$ based on the input $\mathbf{x}_l$ as:

$$\mathbf{x}_{l+1} = \mathbf{x}_l \cdot \sigma(\mathcal{F}(\mathbf{x}_l, \mathbf{W}_l)), \tag{7.4}$$

where $\sigma(x) = \frac{1}{(1+e^{-x})}$, representing the sigmoid function. When recursively applying this building block within an deep architecture, the final output vector $\mathbf{x}_L$ can be obtained as:

$$\mathbf{x}_L = \mathbf{x}_l \cdot \prod_{i=l}^{L} \sigma(\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i)), \tag{7.5}$$

for a network with L sigmoidal blocks. This results in different backward propagation properties in comparison to the original residual building block. Following the chain rule of backpropagation, the backward path can be formulated as:

$$\frac{\partial E}{\partial \mathbf{x}_l} = \frac{\partial E}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial E}{\partial \mathbf{x}_L}(\prod_{i=l}^{L} \sigma(\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i)) + \mathbf{x}_l \frac{\partial}{\partial \mathbf{x}_l} \prod_{i=l}^{L} \sigma(\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i))). \tag{7.6}$$

While the gradient of the original residual block obtains one term that directly propagates information without any interference of other layers, the gradient of the sigmoidal block is interfered by the term $\prod_{i=l}^{L} \sigma(\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i))$. Without sigmoid function, large values of $\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i) > 1$ result into exploding gradients for deep architectures. However, through applying the sigmoid function before multiplication, the values of the term are constrained to $0 < \sigma(\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i)) < 1$, which consequently resolve exploding gradients. On the other hand, small values of the term $\sigma(\mathcal{F}(\mathbf{x}_i, \mathbf{W}_i)) < 0$, in combination with very deep architectures results into vanishing gradients due to the multiplication. The experiments done with the sigmoidal block, however, indicate that this issue is not arising for commonly used depth $L < 20$, and the final performance is not inferior to residual blocks. Thus, while the scalability of the proposed block is in principle problematic, it is a necessary trade-off for a larger design space in pruning based architecture search.

7.4 Structure definitions

Standard machine learning frameworks, libraries and compilers, such as TensorFlow, cuDNN, TensorRT, etc. constrain their users by only enabling certain configurations of operators. The design parameters for convolution operators allow arbitrary filter width and height, striding, group size and number of input or output channels. All of these configurations must be inline with the constraints of

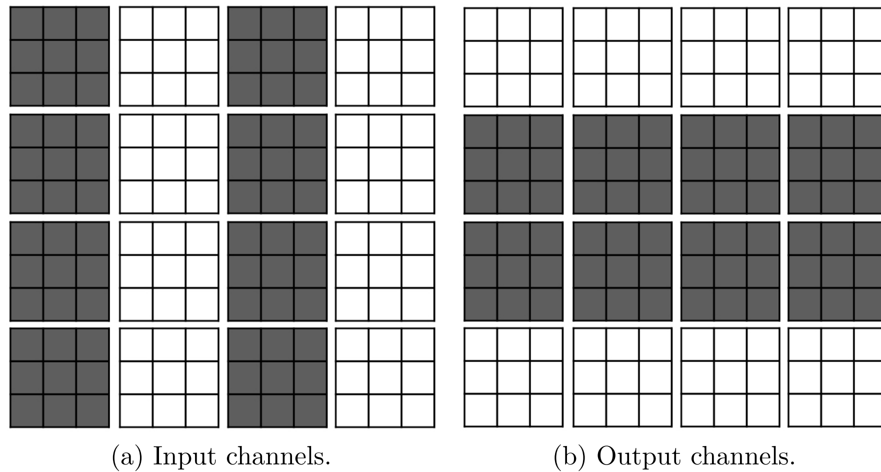


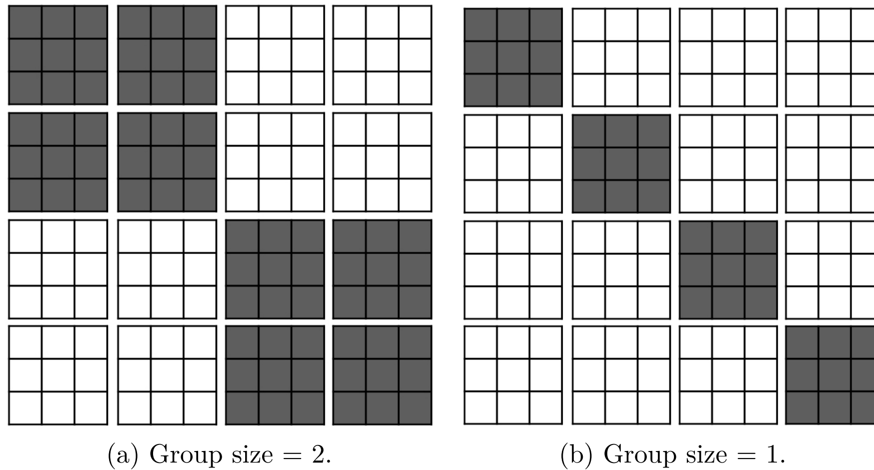
Fig. 7.8 Channel pruning within a convolution operator.

their respective building blocks. Most of the structures definitions in Section 6.4 either violate or do not cover the design parameters of such frameworks, with the exception of channel and layer pruning. This section explains the required extensions of PSP in order to perform extensive architecture search through pruning.

7.4.1 Channel pruning

The most important structure for architecture search is pruning of channels. Figure 7.8 illustrates the four-dimensional convolution tensor, where the large squares represent the kernels, and the corresponding horizontal and vertical dimensions represent the number of input and output feature maps, respectively. While pruning input channels (Figure 7.8a) is already explored in Section 6.4, output channels (Figure 7.8b) are of utmost importance for the previously introduced building blocks. The pruning is implemented - following the PSP technique - by adding auxiliary parameters for each input or output channel to either weight or activation tensor.

For the Inception block, output channel pruning is used for obtaining $w_{i,1}$, $w_{i,2}$, $w_{i,3}$ and the resulting $w_{i,0}$. For residual and sigmoidal blocks, pruning of input channels is used for $w_{i,0}$ and $w_{i,1}$, while output channels are required for w_s. Last, the dense block requires output channel pruning for $w_{i,1}$, $w_{i,2}$ and $w_{i,3}$. Both pruning structures are highly desirable because they do not only reduce parameters and computations, but also activations and, therefore, the overall



(a) Group size = 2. (b) Group size = 1.

Fig. 7.9 Group pruning within a convolution operator.

memory requirements.

7.4.2 Group pruning

Grouping of convolution operations refers to splitting the feature maps and performing independent computations on them in order to reduce the requirements of a model. Figure 7.9 illustrates group pruning on the example of a group size of two (Figure 7.9a) and one (Figure 7.9b). This pruning structure is realized by iteratively splitting the weight tensor and parameterizing the diagonal elements of the splits. Consequently, the group size g_i of a layer is determined by the number of input/output feature maps w_i and the split s_i which is learned through pruning, as: $g_i = \frac{w_i}{s_i}$.

Following this technique, the possible splits per layer are constrained to $s_i = 0, 1, 2, 4, 8, 16$ or 32, in order to reduce training time. In comparison to other techniques, this enables a much larger design space. Related publications on the subject of architecture search (either automatic or hand-tuned) highly leverage group convolutions by either setting the group size or split to a constant value across the whole network. While the vast majority of related work uses $g_i = 1$ (also denoted as depth-wise convolution) to reduce the possible search space, recent work [68] showed that $g_i > 1$ achieves better efficiency. The main benefit of such grouping techniques is that they allow extremely parameter and computation efficient models while achieving high accuracy. However, they often result in tremendous activation requirements, resulting in a high number of

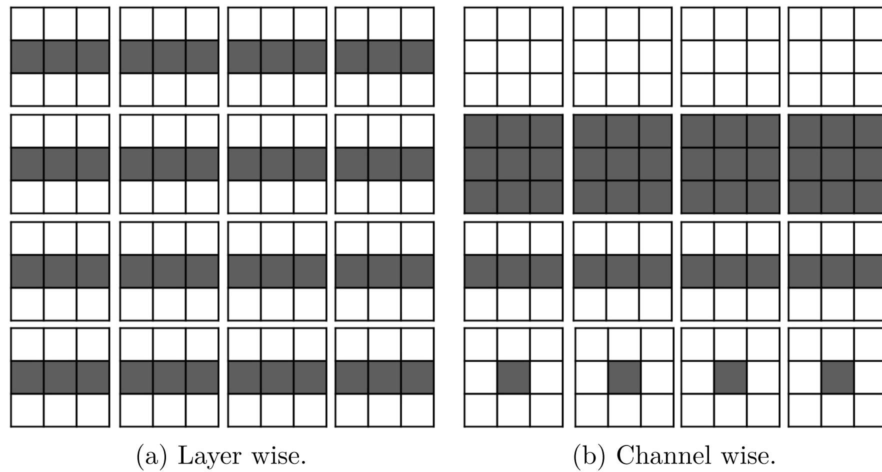


Fig. 7.10 Kernel pruning of a convolution operator.

memory accesses and poor data reuse and, thus, poor inference performance.

7.4.3 Kernel pruning

The kernel size $k_{w,i} \times k_{h,i}$ per layer i is one of the most critical design parameters of convolution layers, because large kernels result in high resource requirements while small kernels may not be sufficient for detecting longer distance correlations. Kernel pruning (see Figure 6.2d) is already introduced in Section 6.4, however, it is not applicable to the targeted compute stack as it only allows dense kernels. In order to generate kernel pruning in a way that it is inline with libraries and compilers, the structure definition needs to be able to create dense kernels with variable width $k_{w,i}$ and height $k_{h,i}$. This pruning variant is implemented through a parameterization of horizontal and vertical slices for all kernels (Figure 7.10a) or per output channel (Figure 7.10b) of a layer.

If the pruning is performed for all kernels of a layer (Figure 7.10a), any height and width combination is possible $k_{w,i}, k_{h,i} \geq 0$, which implicitly includes layer pruning. If it is performed per output channel (Figure 7.10a), only quadratic kernels with odd configurations are enabled, in order to reduce the amount of parallel executed layers. The latter techniques is used to generate Inception like architectures (see Figure 7.1) from a single dense layer, which allows better performance through higher utilization and larger models. Related architecture search techniques only employ squared kernels with odd configurations: $k_{w,i} = k_{h,i} = 1, 3, 5, 7$, ultimately limiting the possible design space significantly. On

the contrary, learning the kernels through pruning enables a much larger design space and, therefore, better efficiency for the resulting inference models.

7.4.4 Combining pruning methods

In order to increase the overall efficiency, it is necessary to combine different pruning structures which is not trivial with respect to the software stack. Kernel pruning can be simply combined with channel as well as group pruning while being compatible with the targeted software stack. However, the combination of channel and group pruning interfere each other, preventing these structures to be applied together dynamically. The main problem here is that all groups within a layer are required to have the identical amount of feature maps and channel pruning potentially produces varying amounts of feature maps. In order to resolve this issue, the group size g_i can be set statically for each layer and channel pruning applied dynamically during training. Here, it is necessary to share the same pruning mask across all groups within a layer to guarantee that all groups obtain the same amount of feature maps.

7.5 Elastic-net regularization

The initially introduced technique for PSP leverages ℓ_2 regularization with a fixed regularization strength λ to force the structure parameters towards zero and uses the threshold parameter ϵ (see Equation 5.2) to create sparsity. This combination aims to remove as much unimportant structures during training as possible while not sacrificing prediction performance. However, the structure definitions from Section 6.4 are much finer than the required structures for architecture search and, hence, allow higher sparsity. More coarse-grained structures, such as groups or kernels, require heavier regularization for creating sparsity which cannot be simply realized by increasing the regularization strength of the ℓ_2 penalty or threshold parameter ϵ.

To create high sparsity ratios for these structures, ℓ_1 regularization needs to be considered. Section 6.5.1 empirically compares ℓ_1 and ℓ_2 regularization - in terms of accuracy with respect to sparsity - and indicates that the latter performs significantly better. Figure 6.1 shows that ℓ_1 regularized parameter distributions are roughly one order of magnitude larger than equally trained

networks with ℓ_2 regularization. Thus, the effective learning rate is lower for ℓ_1 regularized parameter which ultimately reduces the generalization performance of the model. Consequently, the regularization term needs to be a combination of ℓ_1 and ℓ_2 as:

$$E_{\ell_e}(\alpha_i) = E(\alpha_i) + \lambda_{\ell_1}|\alpha_i| + \frac{\lambda_{\ell_2}}{2}\alpha_i^2, \tag{7.7}$$

where λ_{ℓ_1} and λ_{ℓ_2} are the respective regularization strength parameters. This technique is also denoted as *Elastic-net regularization* in related literature [84]. Based on this modification of the loss function, the updates for the structure parameters are calculated as:

$$\Delta\alpha_i(t+1) = -\eta\frac{\partial E}{\partial \alpha_i} - \lambda_{\ell_1}\eta\,\mathrm{sign}(\alpha_i) - \lambda_{\ell_2}\eta\alpha_i. \tag{7.8}$$

As can be seen, the $-\lambda_{\ell_2}\eta\alpha_i$ term pushes the parameters towards zero and enables better generalization through high effective learning rate, while the $-\lambda_{\ell_1}\eta\,\mathrm{sign}(\alpha_i)$ term pushes certain structures to zero and creates sparsity. For the original PSP training, λ_{ℓ_2} is set to a fixed value and pruning threshold ϵ is increased for more sparsity. The training setup is changed here by setting λ_{ℓ_2} and ϵ to a fixed value and varying λ_{ℓ_1} in order to control the sparsity.

7.6 Evaluation

The evaluation of previously elaborated building blocks and structures is performed on the CIFAR-10 as well as ImageNet task and analysed with respect to performance. The performance is measured using theoretical metrics such as number of MACs, parameters, activations and timing metrics for inference. Latency and throughput (in FPS) are evaluated using TensorRT and cuDNN on NVidias Jetson Nano and Xavier boards. All training runs use a fixed value for the ℓ_2 regularization strength as $\lambda_{\ell_2} = 10^{-4}$ while ℓ_1 is varied according to the desired amount of sparsity. The threshold parameter for pruning is set to $\epsilon = 0.01$ for all experiments.

7.6.1 CIFAR-10

This section explores the performance of different building blocks as well as pruning structures on the CIFAR-10 task. Figure 7.11 reports the theoretical

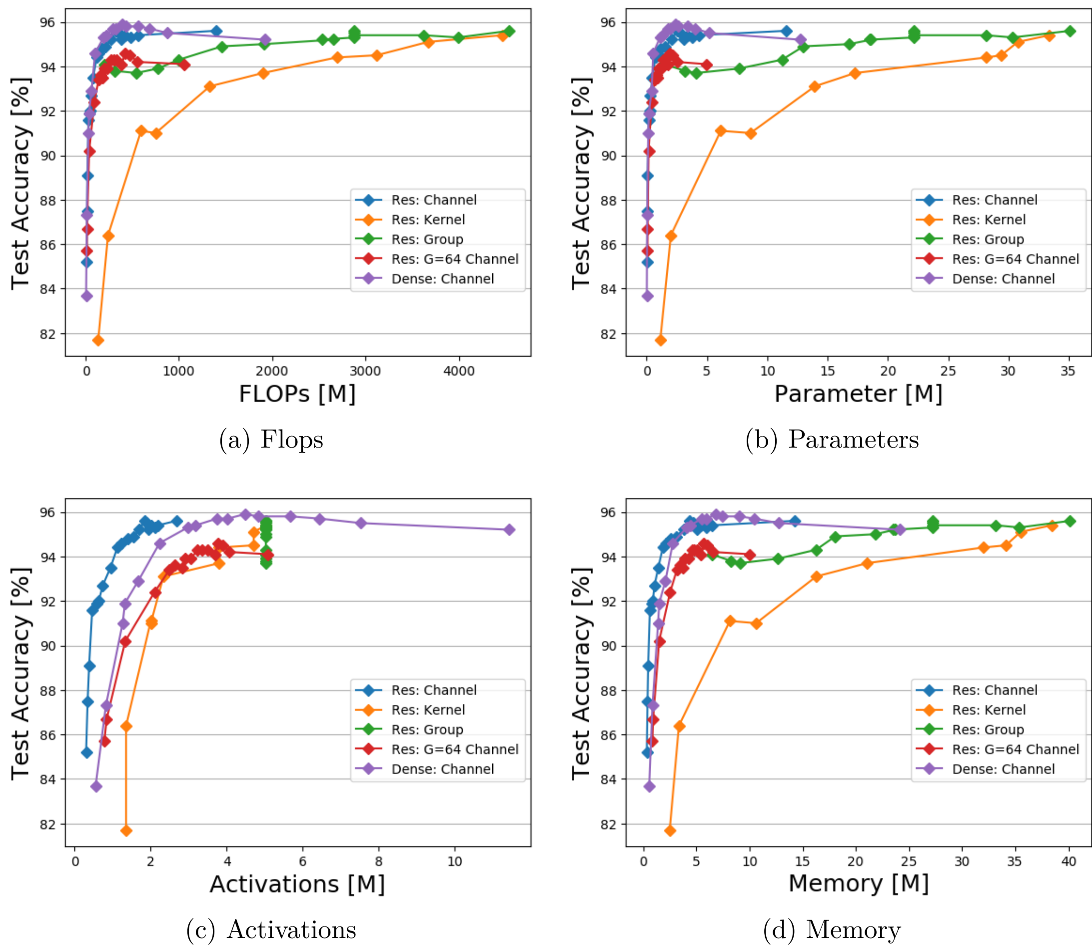
and Figure 7.12 measured metrics of the evaluation.

Comparing Different Structures

The initial configurations of the neural networks follows the Wide Residual Networks (WRN) [85] with a depth of 28 layers, as this is one of the top performing architectures on this task. WRN uses the basic residual block (see Figure 5.2) with 3×3 convolutions and without grouping. The WRN architecture is modified to use the sigmoidal residuals (see Figure 7.7) in order to enable the pruning of feature maps. It should be noted that this modification does not result into accuracy degradation in comparison to the original residual block.

In a first step channel (*Res: Channel*), group (*Res: Group*) and kernel (*Res: Kernel*) pruning are evaluated separately on the modified WRN architecture. When considering FLOPs and parameters, which are the main metrics in related literature with respect to resource-efficient neural networks, it is clear that channel and group pruning significantly outperform kernel pruning. This indicates a high sensitivity of the kernel size to the overall accuracy. Group and channel pruning perform very similar on these two metrics, especially for highly compressed models. However, when also considering the activation and overall memory (parameters + activations), group pruning performs significantly worse because it does not remove any activations and relies on a well-defined initial configurations. As a result, channel pruning is the best performing compression structure when applied in isolation.

In a next step, the group size is set to $g_i = 64$ and channel pruning is applied (*Res: G=64 Channel*), in order to evaluate the performance of combined sparse structures. This combination performs worse than pure channel pruning for all metrics and requires a large amount of activations. Ultimately, it can be stated that group convolutions are excellent at reducing FLOPs and parameters but can harm the overall memory requirements by increasing the amount of activations. This observation is in clear contrast to related literature where group convolutions are heavily applied ($g_i = 1$) on this task and activations (or memory) are basically ignored.



(a) Flops

(b) Parameters

(c) Activations

(d) Memory

Fig. 7.11 Performance metrics of several pruning structures and building blocks.

Building Blocks

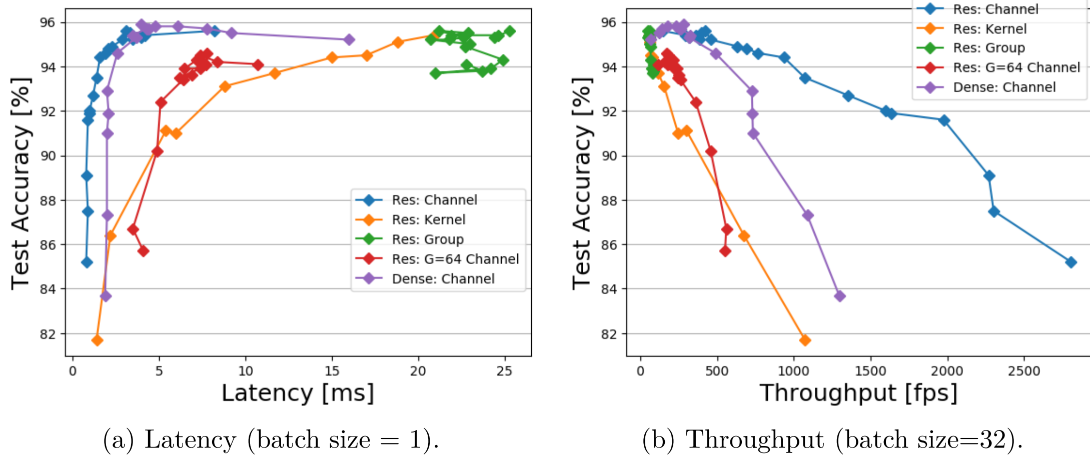
Previous experiments found that pure channel pruning performs best on CIFAR-10 task. This section compares the performance of channel pruning on (sigmoidal) residual and dense building blocks, in order to find the best performing architecture and compression setting on this task. A DenseNet variant is created for this experiment which uses the basic dense block (see Figure 7.4). The architecture is scaled in depth to 28 layers and the width is varied until it roughly matches parameters and computations of the (sigmoidal) residual architecture from previous section to guarantee a fair comparison. No group convolution is applied and the channels are removed dynamically through pruning. Figure 7.11 compares both architecture using residual (*Res: Channel*) and dense (*Dense: Channel*) building blocks.

As can be seen, the dense outperforms the residual block in terms of FLOPs as well as parameters. In terms of activations, however, residual blocks are clearly more beneficial, which also influences the overall memory. In summary it can be stated that dense building blocks are more parameter/computation efficient and residual blocks are more memory efficient.

Measuring Inference Speed

The previous section explored several pruning structures and building blocks which indicate different potential with respect to computation and memory efficiency. This section evaluates how these metrics impact the inference speed in terms of latency (with a batch size of 1) as well as throughput (batch size of 32). Figure 7.12 reports the inference metrics for the Jetson Nano board using half-precision floating point for weights and activations.

As can be seen, the various models in each regime (structure or building block) show similar behaviour for latency and throughput. The worst performing regimes are group and kernel pruning as well as the combination of fixed grouping and channel pruning. Especially interesting is group pruning: although it greatly reduces FLOPs and parameters, it fails at translating this reduction into faster computations. As opposed to this, pure channel pruning using residual and dense building blocks achieves the best performance. These results highlight the importance of reducing memory (or more specifically activations) rather than FLOPs, in order to reduce latency or increase throughput.



(a) Latency (batch size = 1). (b) Throughput (batch size=32).

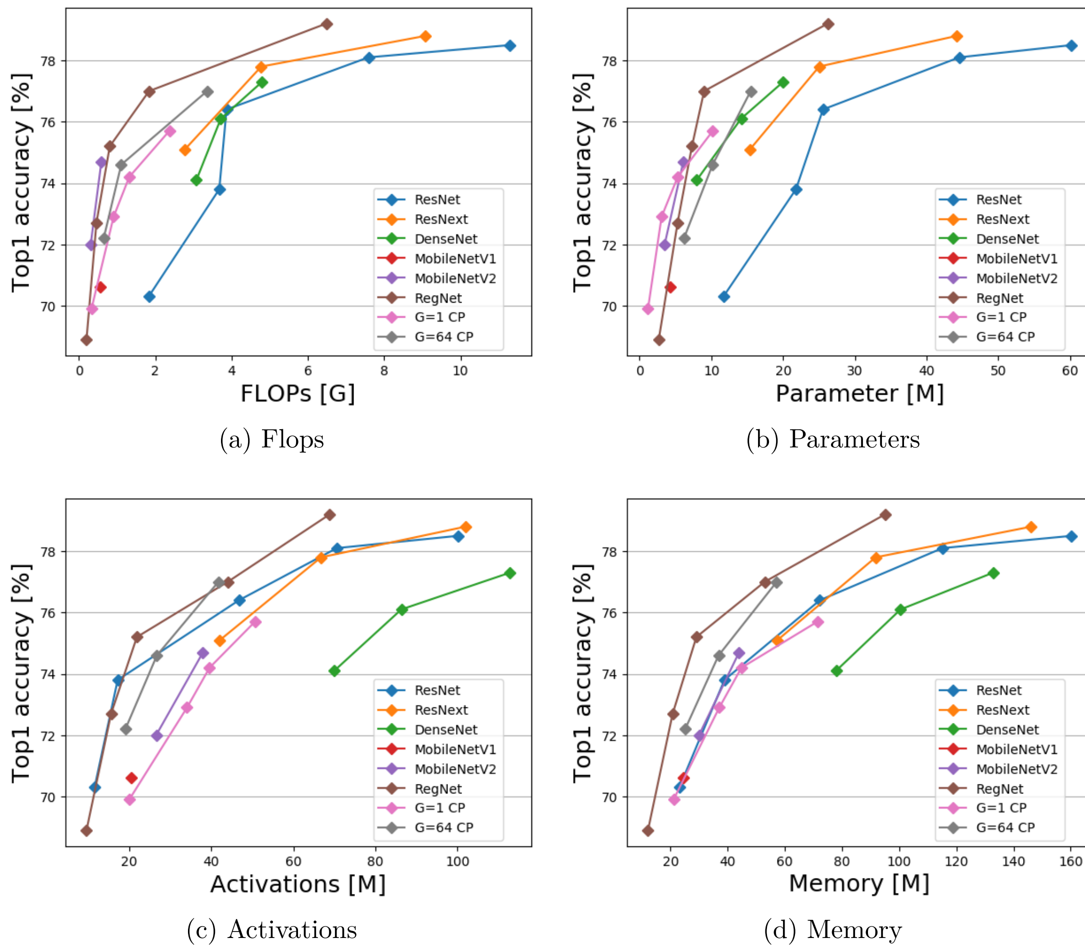
Fig. 7.12 Inference metrics of several pruning structures and building blocks.

7.6.2 ImageNet

This section evaluates several promising models as well as pruning on the large-scale ImageNet task. The evaluated models in this section are the most popular architectures proposed in related literature: ResNet [14], ResNext [67], DenseNet [15], MobileNetV1 [65], MobileNetV2 [66] and RegNet [68]. The reported accuracy of these models is extracted from the original publications while the pruning results are obtained through training the models. Please note that only models are evaluated which share a similar training setup (i.e. number of epochs, data augmentation, stochastic regularization) and convolution layer only (which excludes models with squeeze-and-excitation modules) to guarantee a fair comparison. Furthermore, all models are rebuilt and evaluated on hardware using an identical benchmark setting.

Comparing Different Models and Structures

The performance metrics in terms of FLOPs, parameters, activations and memory of the above mentioned models are reported in Figure 7.13. Models without group convolutions (ResNet, DenseNet) or with a constant split rather than group size (ResNext) are the worst performing in terms of FLOPs and parameters. In contrast, models with constant group sizes such as MobileNetV1 ($g = 1$), MobileNetV2 ($g = 1$) and RegNet ($g = 8, 16, 24, 48$) perform best on these metrics. In terms of activations, however, models with $g = 1$ perform worse than those with larger or no group size. Interestingly, almost all models have similar overall



(a) Flops (b) Parameters

(c) Activations (d) Memory

Fig. 7.13 Performance metrics of several architectures.

memory requirements, with the exception of RegNet (best performing) and DenseNet (worst performing).

This analysis indicates that the overall best performing architectures - in terms of compute and memory efficiency - are those with a constant group size (MobileNetV1, MobileNetV2 and RegNet). Furthermore, it can be assumed that larger group sizes ($g > 1$) are the only configurations that allow for a better memory efficiency. However, this assumption cannot be verified by the available data because the RegNet models are optimal representations for this task, extracted from distributions of hundreds of models, while the MobileNets are found using rather simple heuristics. As a consequence, it is possible that RegNet performs better because of much more training time invested in finding well-performing architectures.

In order to effectively evaluate the impact of the group size on accuracy

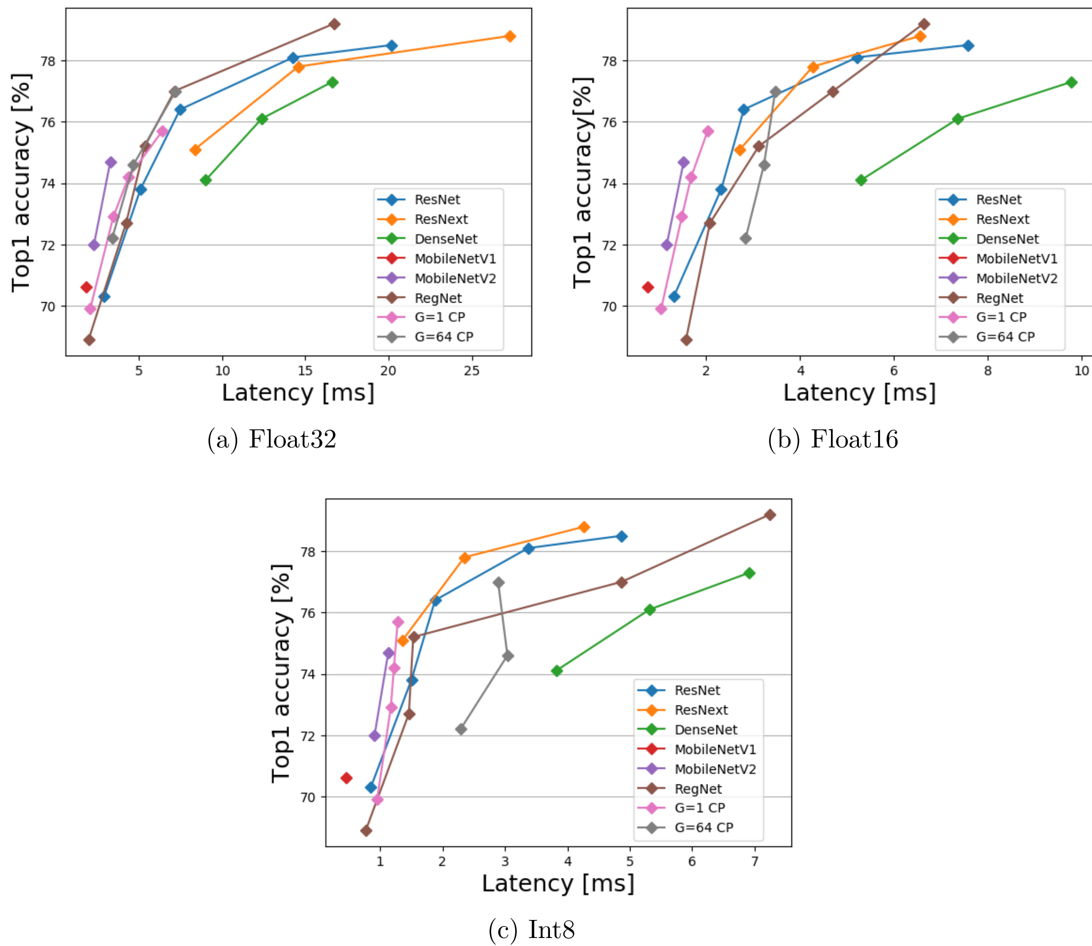
and memory efficiency, an isolated experiment is necessary. For this evaluation, two models with equal configuration but different group sizes are created and compressed using channel pruning. The model configuration used here follows the ResNet-50 architecture with 3 - 4 - 6 - 3 blocks in their respective stages (56×56 - 28×28 - 14×14 - 7×7). The building blocks are implemented using sigmoidal residuals without bottleneck, since compression is achieved through grouping and channel pruning. Figure 7.13 reports the metrics of both models, where one model uses a group size of $g = 1$ (*G = 1 CP*) while the other uses a $g = 64$ (*G = 64 CP*) for the inner convolution layer.

In terms of FLOPs, both models show a similar efficiency whereas the $g = 1$ configuration achieves a better parameter efficiency. With respect to activations, however, the $g = 64$ configuration is significantly more efficient which ultimately translates into a better overall memory efficiency. Interestingly, the memory consumption of the $g = 1$ configuration is almost identical to ResNet, ResNext, MobileNetV1 and MobileNetV2. These results validate that group convolutions are necessary to reduce computations on this task but only larger group sizes are able to reduce memory requirements. Another promising observation is the effectiveness of the proposed pruning methodology: both evaluated models show competitive performance to their MobileNet and RegNet counterparts but require much less training time to develop.

Measuring Latency

This section studies the impact of the various analyzed models from the previous section on the inference time using half- and single-precision floating point as well as 8-bit integer representations. Figure 7.14 reports the latency (batch size = 1) of several precision formats on the Jetson Xavier and Figure 7.15 on the Nano board.

As can be seen, the overall best performing architectures for all precision formats on the Xavier board are architecture using residual blocks with a group size of $g = 1$. This is surprising since the theoretical analysis showed that larger groups are equally compute and even more memory efficient. In general, there can be an increasing performance gap observed with decreasing precision formats between models with $g = 1$ and $g > 1$. This gap cannot be explained by some principled issues but rather by certain practical considerations: virtually any work on architecture search focuses on models using $g = 1$, due to their extreme



(a) Float32

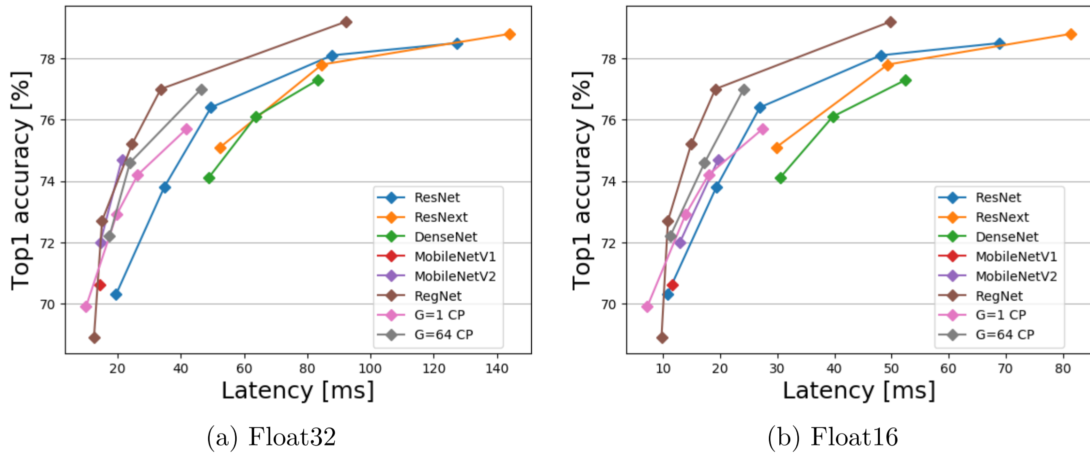
(b) Float16

(c) Int8

Fig. 7.14 Latency on Jetson Xavier.

efficiency with regard to FLOPs and parameters, which is the main metric studied in related literature. The work done by Radosavovic et al. [68] (introducing the RegNet family), is the first that actually studied larger groups and the impact on activation memory as well. Therefore, hardware vendors optimize their compute stack with respect to highly used configurations and lack the efficient support of novel configurations.

As opposed to the latency results on the Xavier board, the best performing models on the Nano board are configurations with larger groups for all precision formats. These results are inline with the theoretical analyses and highlight the importance of careful architecture selection. However, the gap between models with $g = 1$ and $g > 1$ is lower than expected on the basis of theoretical improvements.



(a) Float32 (b) Float16

Fig. 7.15 Latency on Jetson Nano.

7.6.3 General Feature Scaling

The previous sections extensively studied the impact of various building blocks and sparse structures on compute and memory requirements as well as inference time. These results indicate that residual building blocks achieve superior efficiency in comparison to other techniques. Especially the combination of residual blocks with channel pruning creates the most promising architectures by removing whole feature maps from the model. This section introduces a general pattern of feature-map scaling that frequently appears when channel pruning is applied to models using residual blocks.

Designing neural architectures with residuals requires the engineer to carefully select the number of feature maps in the respective blocks (see Figure 7.3). A common tactic here is to use a constant width multiplier α (typically $\alpha \approx 1.5-2.5$) after each stage, which ultimately scales the amount of feature maps after each stage as $w_s = w_{s-1} \cdot \alpha$. The amount of features for the inner layer are set based on the (inverted) bottleneck ratio β (typically $\beta \approx 0.25-6.0$) as $w_{i,0}, w_{i,1} = \beta \cdot w_s$. For instance, ResNet models usually set $\alpha = 2.0$ and $\beta = 0.25$, which dramatically reduces the search space. On the contrary, channel pruning enables finding fine-grained solutions for each block and, consequently, achieves better efficiency. Figure 7.16 shows the amount of feature maps for each layer and for several training epochs when channel pruning is applied, where each colour denotes a different stage. In this experiment, the regularization strength is increased every 5 epochs. As can be seen, there can be an exponential increase of feature maps

observed, representing a general tendency when pruning such models.

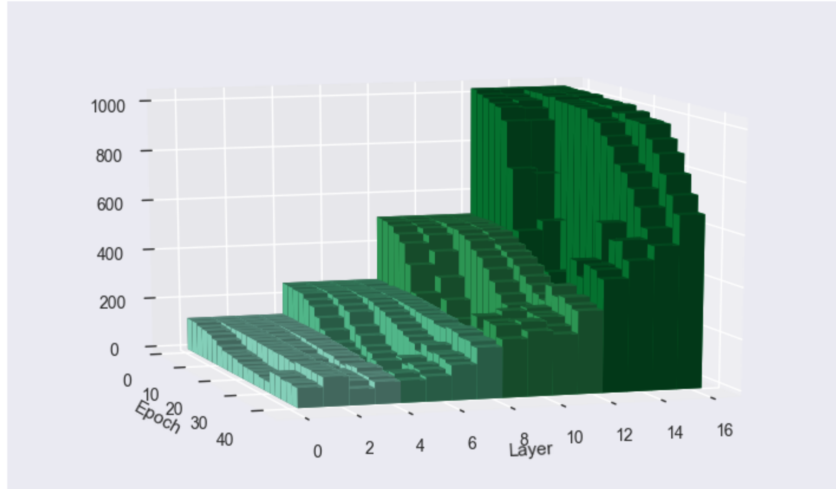


Fig. 7.16 Feature visualization

This observation enables the development of an alternative scaling rule for feature maps, removing the requirement for channel pruning and has a similar search space to a constant width multiplier and (inverted) bottleneck ratio. Let N be the number of layers within a model, then the number of features of layer n can be obtained as: $w_n = a \cdot e^{n \cdot b} + c$, where a, b, c is the design space that can be found by hand or any automatic heuristic technique (i.e. reinforcement learning). Figure 7.17 exemplifies such a scaling rule based on the fit using data obtained from the pruned models (see Figure 7.16).

As can be seen, the resulting model overlaps the hand-designed ResNet in the early layers, but has much more features in the later layers before the feature maps are forwarded to the final prediction part of the model. Such an exponential scaling rule makes intuitively sense, since it can be assumed that every layer adds more features to the model with increasing depth. This is also promising from a computational perspective, because it mainly increases the amount of parameters, which has only a small impact on the overall memory and compute requirements.

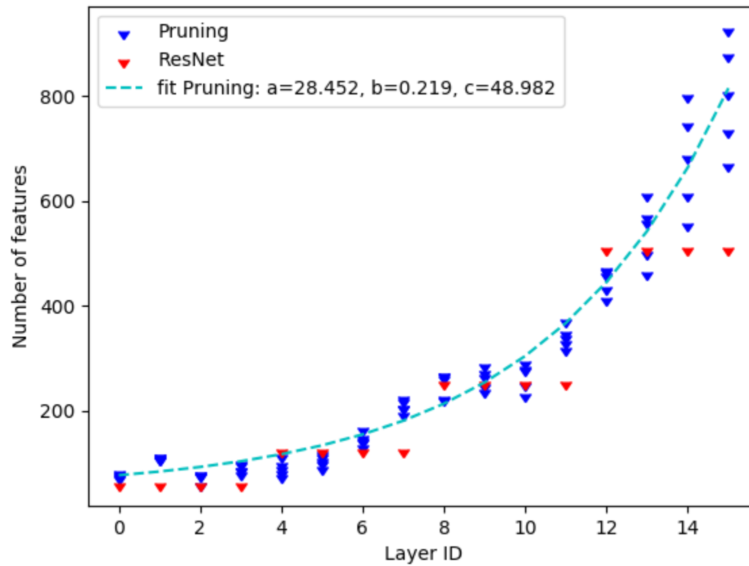


Fig. 7.17 Feature scaling.

7.7 Summary

Designing neural networks by searching for appropriate architectures is one of the key challenges in the field of deep learning. The ultimate goal is to cover as many design parameters as possible while being also efficient in terms of training time and resources. This section proposed a novel combination of structured pruning and efficient building blocks in order to find such architectures. The methodology introduced in this section is beneficial for mainly two reasons: first, it enables to study the impact of building blocks and sparse structures for a certain task by covering a large design space without the need for extreme long training time. Second, it allows to compress neural networks in a structured way, which is inline with virtually any accelerator and their respective software stack. The key insights of this sections are:

- Inference time in terms of latency and throughput mainly correlate to memory requirements rather than computations and parameters.
- The pervasively used depth-wise convolution (group size of 1) is excellent at reducing computations and parameters, but fails at reducing the number of activations, leading to poor inference and training performance.

- The best performing models on the popular CIFAR-10 and ImageNet tasks use a combination of residual blocks without or with large group sizes.

Chapter 8

Comparing Compression Techniques on Hardware

The last chapters studied several compression techniques with considerations on the targeted software stack and hardware. Each chapter has a focus on leveraging compression to accelerate inference computations as much as possible while maintaining prediction quality of the uncompressed model. While baseline models (i.e. real-valued operators or un-pruned tensors) reveal insights on how much performance can be improved for a certain processor, there is no absolute overview of the various hardware regimes. This chapter compares these specialized forms of compression on their respective hardware in terms of absolute performance to evaluate the most promising compute concepts for neural networks.

The comparison is done using small-sized and resource-constrained embedded processors which exhibit a similar energy regime. An ARM Cortex-A57 is used as representative for CPUs, NVidia's Jetson Nano for GPUs and the Xilinx Ultra96 for FPGAs. CIFAR-10 is chosen for this comparison as it offers a good trade-off between real-world example and training time, which is essential for the viability of such an evaluation. ResNet variants are used for CPU as well as GPU experiments and a VGG variant for FPGA, sine the used FINN framework does not support residual connections. Figure 8.1 reports the performance in terms of FPS over accuracy of the various models and the three devices.

The reported metrics for CPU inference correspond to 8-bit integer quantization in combination with the Gemmlowp library, sparse-ternary quantization using the RaS algorithm and binarized weights/activations using the binary

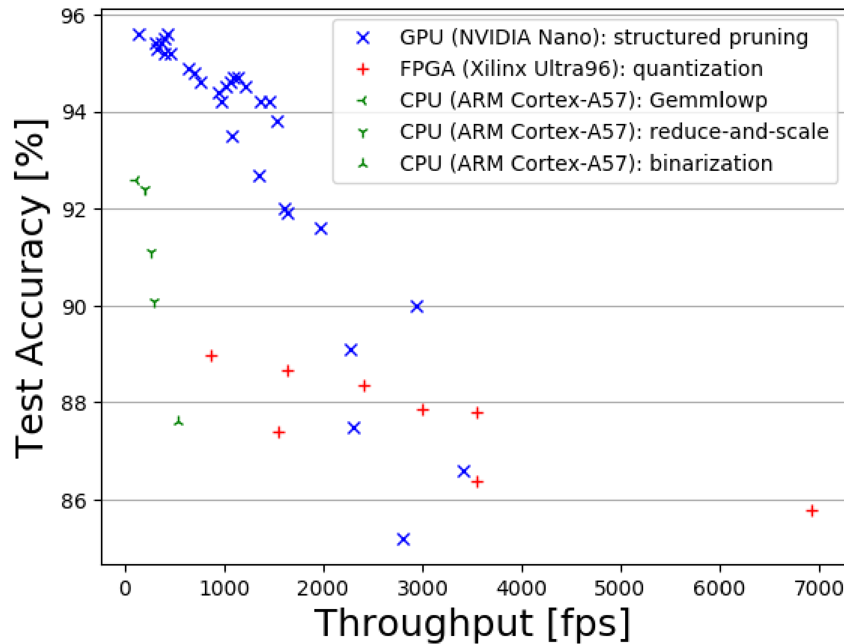


Fig. 8.1 Overall comparison of several compression techniques on CPU, GPU and FPGA using the CIFAR-10 dataset.

Eigen extension. This variety of different compression and algorithm techniques highlights the flexibility of such processors and the potential to leverage compression for inference. ARM's ISA features SIMD instructions for bit manipulation and integer formats, enabling low-precision computations while also supporting floating-point arithmetic. Furthermore, the combination of high-frequency and low-parallelism cores enables the deployment of unstructured sparsity. This means that CPUs are well suited for compressed neural networks, however, the comparison to massively-parallel accelerators shows that they lack the necessary amount of parallelism to achieve competitive throughput. The upside of CPUs is that they feature a relatively large memory, allowing large and accurate models to be deployed.

The reported metrics for FPGA inference correspond to uniformly quantized weights/activations together with bit-serial inference using the FINN framework. It is possible to implement dedicated hardware instances for each layer and perform inference in a pipelined fashion through re-configurable logic, which virtually avoids off-chip memory accesses. Furthermore, bit-serial processing elements require only few logic instances in comparison to their floating-point and integer counterparts. This enable FPGAs to excel at extreme high-throughput

inference and high utilization of the available hardware resources. Such data-flow architecture, however, demand the entire model (including activations) to stay on chip, possibly preventing larger models to be deployed.

The reported metric for GPU inference correspond to structured sparse models using channel pruning in combination with half-precision floating point formats. GPUs are relatively constrained in terms of flexibility when deploying compressed models, due to the requirement of using optimized libraries and their respective software stack. However, they show a good compromise of general-purpose and re-configurable processor, enabling high throughput and accuracy. Additionally, they feature a large off-chip memory which allows - together with latency hiding techniques - high-throughput inference of large models.

Chapter 9

Discussion

In this chapter, the main insights from this work are reviewed and further discussed, especially in a more global context. Furthermore, current trends in technology are taken into account which have been proposed in related literature while the work on this thesis was done. Finally the broader impact of this work is evaluated and its potential use towards further research directions.

9.1 Potential and Limitations of Compression

Compression techniques for neural networks are an excellent tool for creating efficient inference models for resource-constrained systems. This section reviews the potential of the analyzed techniques as well as their implications and limitations with respect to task and hardware.

9.1.1 Data Representation

The insights gained in this work highlight the potential of quantization while also considering the implications with respect to model accuracy and hardware platform. Binarization greatly reduces memory and the resulting computations suit computer architectures very well, however, severe accuracy degradation is likely to occur. Other forms of extreme quantization can avoid the accuracy loss but are difficult to realize on hardware due to different bit requirements. For instance, the evaluations indicate that activations require more bits than parameters and LSTM require more bits than convolution or fully-connected layers. This generates a high variance in possible bit-combinations.

Supporting any bit-combinations of weights and activations using integer arithmetic units is infeasible in hardware due to area constraints. Bit-serial units can resolve this issue by sequentially computing arbitrary bit-combination, however, at the cost of higher latency caused by serialization. Ras inference might be a good solution for this problem, but scales poorly to massive amounts of parallelism due to load imbalance.

Another concerning implication of quantization is the required hardware resources of the real-valued model, which is required for quantization. While this is not problematic for small models on simple tasks, the required real-valued model is likely to be the limiting factor when targeting large models on complex tasks.

One promising way to resolve the implications of quantization is to use rather conservative forms of bit representations. The community standard for such conservative quantization forms is currently half-precision floating point for training and 8-bit integer for inference. The recently proposed Posit [86] format might be a good solution for reducing the bit with for both, training and inference.

9.1.2 Structural Efficiency

Structural efficiency through structured pruning or pre-defined structured sparsity (i.e. architecture search) is likely to be the most promising candidate for efficient neural networks. Unlike unstructured sparsity, such compute structures can be mapped very well on massively-parallel processors and reduce the overhead of indexing non-zero elements. The insights gained in this work indicate that there is a large variety of efficient structures within neural networks, however, they need to be selected carefully with considerations on the hardware/software stack as well as tasks.

For instance, many tool stacks of domain-specific or general-purpose accelerators demand using their respective BLAS libraries for utmost efficiency. These tool stacks are rather restrictive in terms of structure definition and potentially slow down the progress of efficient models. Furthermore, one of the key insights of this work is that inference speed (latency or throughput) depends on memory rather than computations. This is in contrast with most architectures published in related literature, as they heavily leverage extreme group convolutions in

order to reduce computations and parameters. The results of this work indicate that large or no group convolutions - depending on the dataset - achieve best performance in terms of latency/throughput over accuracy, although they are not the most parameter and computation efficient models.

Structured pruning is excellent at generating efficient neural architectures from larger over-parameterized models by exhibiting the most important sub-structures. The main implication here is the size of the unpruned model, which usually requires a large amount of memory and training time. Similar to quantization, this could be the bottleneck when complex tasks are targeted or the accelerators used for training do not feature enough memory.

9.2 Transferability of Insights

Identifying useful techniques for deep learning is extremely difficult due to the large variety of possible solutions and strategies. Considerations in terms of computations or memory with respect to accuracy depends on many aspects, such as application, neural architecture, training setup, etc. The methodology of this work follows a strategy which aims to simplify these aspects. This section discusses the transferability of technique and insights of this work to other task, neural architectures and training setups.

9.2.1 Other Tasks

Most of the evaluation in this work is done on image-classification tasks, such as ImageNet, CIFAR-10/100, SVHN and MNIST. The reason behind the task selection is based on comparability: the vast majority of novel techniques for deep learning (i.e. compression or architecture) are evaluated on these tasks, in order to set a common benchmark when comparing metrics such as inference/-training speed or memory/computations. Furthermore, image classification is one of the key tasks where deep learning excels and significantly outperforms feature-engineering concepts. The major drawback of deep learning here is that it consumes much more resources than such hand-designed classification and, consequently, is a well suiting task when considering resource efficiency.

Another important reason why image classification is essential for evaluation is that such models serve as building blocks for other highly important computer

vision tasks. Object detection is the task of image classification with localization of multiple objects, where bounding boxes need to be predicted as well as the object class within each box. State-of-the-art models on this task implement a bounding box generator (i.e. region proposal network) in combination with a standard classification model (i.e. ResNet or MobileNet) as backbone. The computational heavy lifting and memory requirements of such combined models is caused by the classification model. Consequently, improvements of compression techniques on pure classification tasks directly transfer to object detection tasks. Image segmentation is the task of splitting an image into segments where specific pixels that belong to a certain object need to be identified. This is done by predicting classes for each pixel in an image and, hence, forms again a classification task. However, state-of-the-art models for this task do not simply employ standard classification models, but rather define specialized architectures (i.e. UNet). Thus, it can be assumed that the general trends of this work are transferable since both task are alike.

9.2.2 Other Neural Architectures

The neural architectures used in this work consist of mainly convolution layers for feature extraction and fully-connected layers for classification. Variations of such structures are generated through different scaling (i.e. width, depth, etc.) and connectivity (i.e. residual or dense). However, other forms of neural architecture exist and need to be considered as well.

Recently, channel attention techniques (i.e. squeeze-and-excitation modules) are widely used for computer vision tasks and show great potential. Such attention modules are usually implemented as part of the residual blocks and use additional fully-connected or convolution layers on global metrics, such as the arithmetic mean of each feature map. This aims at increasing model complexity with little impact on parameter and computation requirements. The architectures used in this work do not exploit such channel attention modules, because the focus is on comparability of compression technique rather than achieving state-of-the-art accuracy. However, such modules do not interfere with the concepts and insights of this work and add potential for further improvements.

Another important area of neural architectures are Transformer models [87], which highly depend on the attention mechanism. They were originally introduce

in the field of natural language processing in order to overcome limitations of recurrent architectures and exhibit excellent performance on sequence-to-sequence tasks. While combinations of convolutions and attention layers are already very popular, recent research [88] studied the impact of Transformer architectures for computer vision. Surprisingly they achieve similar accuracy than convolution architectures or even outperform them in terms of parameters and computations. The techniques and insights of this work are in principle also transferable to Transformers, however, the benefits are not obviously predictable: the most time consuming operations in Transformers are tensor splits, concatenations and transposes while matrix multiplications only account for a small fraction [89]. Consequently, reducing computations and memory through compression techniques does not result in similar inference improvements as for convolution architectures.

9.2.3 Other Training Setups

All reported results of experiments in this work follow a specific training setup, which is assumed to be a community standard. For instance, all experiments on the ImageNet dataset are performed using random crop-and-resize technique for data augmentation, no explicit regularization but weight decay and a training routine of 100 epochs. The main advantage of this setup is that it enables a fair comparison between different techniques and architectures without reproducing the results of related work, which would not be feasible considering the training time on such tasks.

Most of the recently reported gains in accuracy for such tasks are based on enhancements to the training setup and regularization techniques. For instance, training techniques for ImageNet include deep supervision, Cutout, DropPath, AutoAugment, etc. which improves validation accuracy dramatically, but results into much longer training time (i.e. ≈ 400 epochs). Such different settings prevent a fair comparison and can lead to false assumption of certain techniques: for instance, the authors of EfficientNet [90] claim 4.9x parameter and 11x computation reductions in comparison to ResNet, however, the training time and setup are completely different. Hence, the results are not comparable and do not reflect to advanced architecture proposals. The aim of this work is to create an comparable overview of compression and hardware techniques rather

than producing state-of-the-art performance and, hence, is not competitive with reported results using such advanced training settings. The techniques and insights gained in this work, however, to not interfere with additional training enhancement and allow for further improvements in future directions.

9.3 Broader Impact and Future Directions

The most promising candidates for mobile DL applications are massively-parallel ASICs, using 8-bit integer formats and structured sparse computations. Chapter 7 evaluated the effectiveness of several popular building blocks and design spaces of convolutions in order to identify efficient inference models as well as design principles. PSP in combination with the methodology introduced in Chapter 7 offers a framework that has more potential than just creating efficient inference models. It can be used to evaluate the effectiveness of other performance improving techniques such as data augmentation, modules, etc. as well. This section highlights the broader impact as well as some possible future directions of the proposed technique and methodology.

9.3.1 Advanced Data Augmentation

Data augmentation is an effective technique for improving the prediction quality of DNNs by applying transformations on the training samples, such as rotations, translations, shears, etc. Such augmentation techniques are obviously highly dependent on the targeted task and training data and usually do not generalize well to other tasks. Most augmentations for a given dataset are manually designed using considerations on potential invariances within the training data. Another emerging field are techniques based on a reinforcement policy that searches a defined design space for the best performing augmentation. All of these techniques evaluate a certain augmentation by only considering a fixed architecture and the increase or decrease in validation quality. However, this does not necessarily result into the most efficient techniques, since a certain augmentation may not increase validation accuracy but into less resources. Consequently better results can be achieved by optimizing augmentation and neural architecture together, which can be done by the methodology in Chapter 7 in combination with PSP.

9.3.2 Advanced Modules

Advanced modules for DNNs are an another emerging techniques which can significantly improve prediction quality. Such modules are commonly added to standard architectures (i.e. ResNet, MobileNet) and include shuffling of feature maps or attention techniques. Especially attention modules are used excessively lately and exist is many different forms and variations. Again, the effectiveness of such modules are evaluated by using a fixed architecture and considering only the increase or decrease in validation quality. Using the proposed methodology enables the evaluation of both, architecture and validation quality together. This is a highly desirable goal, since different modules can lead to different DNN architectures, which can not be identified by only evaluating validation quality.

Chapter 10

Conclusion

DNNs are a key technology nowadays and the main driving factor for many recent performance boosts in AI applications. They prove particularly effective when big amounts of data and ample hardware resources are available. The hardware requirements in terms of memory and compute resources are the limiting factor for their applicability in embedded systems. Such systems are handheld or head-worn devices which characterize the future of consumer and industry products.

Efficient ML models, hardware accelerators and software frameworks are of fundamental importance, in order to enable such applications in mobile device. This work presented and analyzed various techniques to improve these fundamental directions from an holistic view and considers them together instead of separately. From ML perspective, hardware and software stacks are considered for developing compression techniques and architectures. From software perspective, algorithms and libraries are developed and selected to adapt to model and hardware demands. Last, hardware platforms are selected with respect to the requirements of ML models and promising computing concepts (i.e. data-flow or loop-back architecture).

The evaluation of various quantization techniques and data representations showed that binarization greatly reduces memory and computation complexity, however, causes severe accuracy degradation for most tasks. Ternary formats significantly improve upon the accuracy of binarization while also inducing implicitly large amounts of sparsity. Experiments using other precision formats indicate that accuracy is more sensitive to activation than to weight quantization. Furthermore, non-uniformly quantized weights are significantly more accurate

than their uniform counterpart but the implementation difficulty for inference increases. Conservative formats, such as 8-bit integer and half-precision floating point, usually do not result into accuracy degradation.

Unstructured sparsity patterns within DNNs can theoretically achieve extremely low-complex inference models, however, they do not map well onto parallel hardware. The pruning framework PSP is able to adapt to the requirements of (massively) parallel processors and creates structured sparse computation, tailored to a specific hardware or software stack. Various experiments indicate different potential with respect to structure granularity, task and over-parameterization of the model.

Identifying efficient DNN design spaces and principles using the PSP framework and architecture modifications discovered some interesting insights: dense connectivity patterns achieve excellent computation and parameter efficiency, however, require large activation memory and result into a high variance of computation dimensions. Residual connectivity offers a good trade-off between computation/parameter and activation efficiency and achieved highest inference performance throughout this work. In addition, aggressive group convolutions achieved excellent computation/parameter efficiency, however, these metrics do not translate well to latency and throughput, since activation memory is increased. The best performing models use either no or larger groups, which is in contrast to architectures proposed in related literature.

All evaluated hardware platforms exhibit a certain potential with respect to model architecture, data format and computation structure. The limited amount of parallelism in CPUs and their high frequency enables efficient computations of highly sparse models. Furthermore, using the RaS technique enables efficient inference of highly sparse and aggressively quantized models in combination. FPGAs excel through reconfigurable logic, enabling implementations of any data formats and, consequently, suiting aggressively quantized models very well. In addition, they allow for data-flow architecture which reduce off-chip accesses to a minimum and result into high utilization of the processing units. The throughput oriented technology behind GPUs are an excellent fit for DNNs in general and can nowadays leverage structured-sparse computations as well as lower-precision formats. Domain-specific accelerators can be specifically tailored to DNNs in terms of data formats, operations and computational dimensions.

This work contributes a comprehensive study of ML techniques in the domain

of embedded system, considering the entire compute stack. It ranges from high-level optimizations such as architecture search and compression to low-level algorithmic and arithmetic implementations. This makes this work valuable to hardware, software as well as ML engineers and researchers targeting the field of embedded ML. In addition, this work motivates more research with a focus on the entire software/hardware stack, which has been neglected in recent years. Finally, several future directions are discussed which build on top of the gained insights and methodologies discovered in this work.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Dr. Holger Fröning. I have been very fortunate to receive his guidance for the last years and for the opportunity to work on such an interesting research topic. In each stage, Holger gave me excellent advice, most generous support as well as sincere and constructive feedback. I am deeply indebted to him for his continuous positive encouragement and the various opportunities I had during the last years.

I would also like to thank my co-advisor, Professor Dr. Franz Pernkopf. I had the privilege to have access to his professional expertise and invaluable advice. Franz enabled me to collaborate with his group members Dr. Matthias Zöhrer and Wolfgang Roth who contributed through their excellent research to this work. The countless insightful discussions with them helped me bridging the gap between hardware and deep learning research.

It has been a pleasure to work with my colleagues from Holger's *Computing Systems Group*. I would like to thank Andrea Seeger, Dr. Benjamin Klenk, Dr. Alexander Matz, Dr. Felix Zahn, Lorenz Braun, Bernhard Klein and Vahdaneh Kiani for their encouragement and help during the last years. Additionally, I want to thank Andreas Melzer, Kevin Stehle and Hendrik Borras who also contributed to this work with their master theses.

I sincerely thank my family and friends who supported me throughout this exciting journey. Thank you for your patients, moral support and for always believing in me.

I gratefully acknowledge funding by the *German Research Foundation (DFG)* under the project number FR3273/1-1 and the *Austrian Science Fund (FWF)* under the project number I2706-N31. I am also grateful to the *Ruprecht-Karls University Heidelberg* and the *Faculty for Mathematics and Computer Science* for the opportunity to pursue the doctorate and the *Institute of Computer Engineering* for the opportunity to work in such an excellent research environment.

References

[1] G. Schindler, M. Mücke, and H. Fröning, "Linking application description with efficient simd code generation for low-precision signed-integer gemm," in *Euro-Par 2017: Parallel Processing Workshops*, Cham: Springer International Publishing, 2018, pp. 688–699, ISBN: 978-3-319-75178-8 (cit. on pp. 3, 35, 36, 38, 40).

[2] W. Roth, G. Schindler, M. Zöhrer, L. Pfeifenberger, R. Peharz, S. Tschiatschek, H. Fröning, F. Pernkopf, and Z. Ghahramani, "Resource-efficient neural networks for embedded systems," in *ArXiv*, 2020 (cit. on pp. 3, 21, 22, 24, 35).

[3] M. Zöhrer, L. Pfeifenberger, G. Schindler, H. Froning, and F. Pemkopf, "Resource efficient deep eigenvector beamforming," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 3354–3358 (cit. on pp. 3, 35, 53).

[4] G. Schindler, M. Zöhrer, F. Pernkopf, and H. Fröning, "Towards efficient forward propagation on resource-constrained systems," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2018, pp. 426–442 (cit. on pp. 3, 58, 62, 64, 67–70).

[5] G. Schindler, W. Roth, F. Pernkopf, and H. Fröning, "N-ary quantization for cnn model compression and inference acceleration," in *Openreview*, 2018 (cit. on p. 3).

[6] ——, "Parameterized structured pruning for deep neural networks," in *6th International Conference on Machine Learning, Optimization, and Data Science (LOD)*, 2020 (cit. on pp. 3, 86, 89, 90, 94, 96, 98, 99).

[7] K. Stehle, G. Schindler, and H. Fröning, "On the difficulty of designing processor arrays for deep neural networks," in *Workshop on IoT, Edge, and Mobile for Embedded Machine Learning (ITEM), collocated with ECML-PKDD*, 2020 (cit. on pp. 3, 107, 109).

[8] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, 1951 (cit. on p. 10).

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, 2015 (cit. on pp. 10, 76, 77, 80).

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012 (cit. on pp. 11, 30).

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015 (cit. on p. 11).

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015 (cit. on pp. 12, 29).

[13] M. Lin, Q. Chen, and S. Yan, "Network in network," in *International Conference of Learning Represenation (ICLR)*, 2014 (cit. on pp. 12, 29).

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016 (cit. on pp. 12, 29, 77, 121).

[15] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269 (cit. on pp. 12, 29, 49, 121).

[16] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, 2013 (cit. on pp. 13, 24).

[17] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng, "Reading digits in natural images with unsupervised feature learning," *Advances in Neural Information Processing Systems (NIPS)*, 2011 (cit. on p. 14).

[18] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 2012 (cit. on p. 14).

[19] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li, "ImageNet: A large-scale hierarchical image database," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009 (cit. on p. 14).

[20] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, 1974 (cit. on p. 15).

[21] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, 2018 (cit. on p. 19).

[22] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017 (cit. on pp. 19, 47, 51).

[23] M. Höhfeld and S. E. Fahlman, "Learning with limited numerical precision using the cascade-correlation algorithm," *IEEE Trans. Neural Networks*, 1992 (cit. on p. 23).

[24] ——, "Probabilistic rounding in neural network learning with limited precision," *Neurocomputing*, 1992 (cit. on p. 23).

[25] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning (ICML)*, 2015 (cit. on p. 23).

[26] M. Courbariaux, Y. Bengio, and J. David, "Training deep neural networks with low precision multiplications," in *International Conference on Learning Representations (ICLR) Workshop*, vol. abs/1412.7024, 2015 (cit. on p. 23).

[27] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *CoRR*, vol. abs/1510.03009, 2015 (cit. on p. 24).

[28] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning (ICML)*, 2016, pp. 2849–2858 (cit. on p. 24).

[29] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems (NIPS)*, 2015 (cit. on pp. 24, 49).

[30] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016 (cit. on pp. 25, 42, 46, 49).

[31] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016 (cit. on p. 25).

[32] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *International Conference on Learning Representations (ICLR)*, 2017 (cit. on pp. 25, 49, 58, 71, 72, 74, 75, 77).

[33] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *European Conference on Computer Vision (ECCV)*, 2016 (cit. on p. 25).

[34] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," in *Neural Information Processing Systems (NIPS)*, 2017 (cit. on p. 25).

[35] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *CoRR*, vol. abs/1603.01025, 2016 (cit. on p. 25).

[36] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave Gaussian quantization," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017 (cit. on pp. 25, 76).

[37] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “Incremental network quantization: Towards lossless cnns with low-precision weights,” in *International Conference on Learning Representations (ICLR)*, 2017 (cit. on p. 25).

[38] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018 (cit. on p. 25).

[39] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K. Cheng, “Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm,” in *European Conference on Computer Vision (ECCV)*, 2018 (cit. on p. 25).

[40] D. Zhang, J. Yang, D. Ye, and G. Hua, “Lq-nets: Learned quantization for highly accurate and compact deep neural networks,” in *European Conference on Computer Vision (ECCV)*, 2018, pp. 373–390 (cit. on pp. 25, 49, 50).

[41] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, “Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients,” *CoRR*, 2016 (cit. on pp. 26, 48, 49).

[42] S. Wu, G. Li, F. Chen, and L. Shi, “Training and inference with integers in deep neural networks,” in *International Conference on Learning Representations (ICLR)*, 2018 (cit. on p. 26).

[43] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “Haq: Hardware-aware automated quantization with mixed precision,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019 (cit. on p. 26).

[44] S. Uhlich, L. Mauch, F. Cardinaux, K. Yoshiyama, J. A. Garcia, S. Tiedemann, T. Kemp, and A. Nakamura, “Mixed precision dnns: All you need is a good parametrization,” in *International Conference on Learning Representations*, 2020 (cit. on p. 26).

[45] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, “Zeroq: A novel zero shot quantization framework,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020 (cit. on p. 26).

[46] H. Li, S. De, Z. Xu, C. Studer, H. Samet, and T. Goldstein, “Training quantized nets: A deeper understanding,” in *Advances in Neural Information Processing Systems (NIPS)*, 2017 (cit. on p. 26).

[47] A. G. Anderson and C. P. Berg, “The high-dimensional geometry of binary neural networks,” in *International Conference on Learning Representations (ICLR)*, 2018 (cit. on p. 26).

[48] Y. LeCun, J. S. Denker, and S. A. Solla, “Optimal brain damage,” in *Advances in Neural Information Processing Systems (NIPS)*, 1989 (cit. on p. 27).

[49] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems (NIPS)*, 1992 (cit. on p. 27).

[50] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems (NIPS)*, 2015 (cit. on p. 27).

[51] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations (ICLR)*, 2016 (cit. on pp. 27, 30, 70).

[52] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Advances in Neural Information Processing Systems (NIPS)*, 2016 (cit. on p. 27).

[53] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2016 (cit. on p. 27).

[54] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," *CoRR*, vol. abs/1708.06519, 2017. arXiv: 1708.06519. [Online]. Available: http://arxiv.org/abs/1708.06519 (cit. on p. 28).

[55] Z. Huang and N. Wang, "Data-driven sparse structure selection for deep neural networks," in *European Conference on Computer Vision (ECCV)*, 2018 (cit. on p. 28).

[56] A. Gordon, E. Eban, O. Nachum, B. Chen, T. Yang, and E. Choi, "Morphnet: Fast & simple resource-constrained structure learning of deep networks," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018 (cit. on p. 28).

[57] J. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *CoRR*, vol. abs/1707.06342, 2017. arXiv: 1707.06342. [Online]. Available: http://arxiv.org/abs/1707.06342 (cit. on p. 28).

[58] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization," in *International Conference on Learning Representations (ICLR)*, 2018 (cit. on p. 28).

[59] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," in *International Conference on Learning Representations (ICLR)*, 2017 (cit. on p. 28).

[60] Y. Li and S. Ji, "L_0-ARM: Network sparsification via stochastic binary optimization," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2019 (cit. on p. 28).

[61] M. Yin and M. Zhou, “ARM: augment-reinforce-merge gradient for stochastic binary networks,” in *International Conference on Learning Representations (ICLR)*, 2019 (cit. on p. 28).

[62] Y. Aflalo, A. Noy, M. Lin, I. Friedman, and L. Zelnik, *Knapsack pruning with inner distillation*, 2020. arXiv: 2002.08258 [cs.LG] (cit. on p. 28).

[63] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1mb model size,” *CoRR*, vol. abs/1602.07360, 2016 (cit. on p. 29).

[64] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016 (cit. on p. 30).

[65] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, 2017 (cit. on pp. 30, 121).

[66] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018 (cit. on pp. 30, 121).

[67] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017 (cit. on pp. 30, 121).

[68] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, “Designing network design spaces,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020 (cit. on pp. 30, 32, 114, 121, 124).

[69] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018 (cit. on p. 31).

[70] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” in *International Conference on Learning Representations (ICLR)*, 2017 (cit. on p. 31).

[71] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018 (cit. on p. 31).

[72] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, “Mnasnet: Platform-aware neural architecture search for mobile,” 2019 (cit. on p. 32).

[73] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” in *International Conference on Learning Representations (ICLR)*, 2019 (cit. on p. 32).

[74] D. Stamoulis, R. Ding, D. Wang, D. Lymberopoulos, B. Priyantha, J. Liu, and D. Marculescu, “Single-path NAS: designing hardware-efficient convnets in less than 4 hours,” in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2019 (cit. on p. 32).

[75] M. Tan and Q. V. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in *International Conference on Machine Learning (ICML)*, 2019 (cit. on p. 32).

[76] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, “Rethinking the value of network pruning,” in *International Conference on Learning Representations (ICLR)*, 2019 (cit. on p. 33).

[77] ARM, “Cortex-a9 neon media - technical reference manual,” Tech. Rep., 2008 (cit. on p. 44).

[78] J. Barker, R. Marxer, E. Vincent, and S. Watanabe, “The third ’CHiME’ speech separation and recognition challenge: Dataset, task and baselines,” in *IEEE 2015 Automatic Speech Recognition and Understanding Workshop (ASRU)*, 2015 (cit. on p. 53).

[79] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network,” in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016 (cit. on p. 71).

[80] N. V. Smirnov and I. V. Dunin-Barkovskiĭ, *Mathematische Statistik in der Technik:* ser. Mathematik für Naturwissenschaften und Technik. Deutscher Verlag der Wissenschaften, 1963 (cit. on p. 76).

[81] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009 (cit. on p. 77).

[82] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, “Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks,” in *Proceedings of the 23rd International Conference on Machine Learning*, 2006 (cit. on p. 80).

[83] J. Ba, J. Kiros, and G. Hinton, “Layer normalization,” 2016 (cit. on p. 81).

[84] H. Zou and T. Hastie, “Regularization and variable selection via the elastic net (vol b 67, pg 301, 2005),” *Journal of the Royal Statistical Society Series B*, 2005 (cit. on p. 117).

[85] S. Zagoruyko and N. Komodakis, “Wide residual networks,” in *Proceedings of the British Machine Vision Conference (BMVC)*, 2016 (cit. on p. 118).

[86] Gustafson and Yonemoto, “Beating floating point at its own game: Posit arithmetic,” *Supercomput. Front. Innov.: Int. J.*, 2017 (cit. on p. 134).

[87] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems (NIPS)*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017 (cit. on p. 136).

[88] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An image is worth 16x16 words: Transformers for image recognition at scale,” in *International Conference on Learning Representations*, 2021 (cit. on p. 137).

[89] H. Wang, Z. Zhang, and S. Han, “Spatten: Efficient sparse attention architecture with cascade token and head pruning,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2020 (cit. on p. 137).

[90] M. Tan and Q. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in *International Conference on Machine Learning (ICML)*, 2019 (cit. on p. 137).