Technische Universität Berlin

Finding Storage- and Compute-Efficient Convolutional Neural Networks

Author:
Daniel Becking
Matriculation Number: 382772

Supervisors:
Prof. Dr. Klaus-Robert Müller
Prof. Dr.-Ing. Thomas Wiegand
Simon Wiedemann

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science
in
Biomedical Engineering

at the Institute of Software Engineering and
Theoretical Computer Science

March 3, 2020

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only sources and resources listed were used.

Berlin, 15.03.2020

. . . . . .

(Signature Daniel Becking)

Abstract

Convolutional neural networks (CNNs) have taken the spotlight in a variety of machine learning applications. To reach the desired performance, CNNs became increasingly deeper and larger which goes along with a tremendous amount of power and storage requirements. Such increases in computational demands and memory make deep learning prohibitive for resource-constrained hardware platforms such as mobile devices. In order to address this problem, we provide a general framework which renders efficient CNN representations that solve given classification tasks to specified quality levels. More precisely, the framework yields sparse and ternary neural networks, i.e. networks with many parameters set to zero and the non-zero parameters quantized from 32 bit to 2 bit. Ternary networks are not only efficient in terms of storage but also in terms of computational complexity. By explicitly boosting sparsity we reach further efficiency gains. The proposed framework follows a two-step paradigm. First, a baseline model is extended by compound model scaling until a specified target accuracy is reached. Secondly, our Entropy-Constrained Trained Ternarization (EC2T) algorithm is applied which simultaneously quantizes and sparsifies the scaled model. Here, a λ-operator balances the entropy constraint and thus the compression gain of the resulting network. We validated the effectiveness of EC2T in a variety of experiments. This includes CIFAR-10, CIFAR-100 and ImageNet classification tasks and the compression of renowned architectures (i.e., ResNets and EfficientNet) as well as our own compound-scaled models. We show the advantages of EC2T compared to the standard in ternary quantization, Trained Ternary Quantization (TTQ), and set new benchmarks in this research area. For instance, EC2T compresses ResNet-20 by more than 24$\times$ and reduces the number of arithmetical operations by more than 12$\times$ while causing a minimal accuracy degradation of 0.9%.

Kurzfassung

Convolutional Neural Networks (CNNs) finden in vielen Bereichen des maschinellen Lernens erfolgreich Anwendung. Um die Leistungsfähigkeit von CNNs zu verbessern, werden diese zunehmend größer, tiefer und komplexer, was mit einem erhöhten Energie- und Speicherbedarf einhergeht. Dieser Bedarf und die benötigte Rechenkapazität sind restriktiv für die Anwendung von Deep Learning auf Geräten mit limitierten Ressourcen, etwa mobilen Endgeräten. Vor diesem Hintergrund wird im Rahmen der vorliegenden Arbeit eine generelle Methode zur Erzeugung effizienter CNNs bereitgestellt, welche vorgegebene Klassifizierungsaufgaben mit definierter Genauigkeit lösen. Die erzeugten neuronalen Netze sind spärlich und ternär, das heißt, viele Netzwerkparameter werden auf Null gesetzt und die verbleibenden Parameter von 32 bit auf 2 bit quantisiert. Ternäre Netzwerke sind nicht nur hinsichtlich ihrer Speichurung effizient, sondern auch hinsichtlich der Rechenkomplexizität. Indem die vorgeschlagene Methode explizit Spärlichkeit im Netzwerk erhöht, können weitere Effizienzsteigerungen erzielt werden. Die Methode folgt einem zweistufigen Paradigma: zunächst wird ein Ausgangsmodell mit einem Verfahren der Modellskalierung erweitert, bis es die gewünschte Genauigkeit erreicht hat. Anschließend wird der Hauptalgorithmus angewandt, Entropy-Constrained Trained Ternarization (EC2T). Dieser quantisiert das CNN während gleichzeitig eine hohe Spärlickeit erzeugt wird. Dabei regelt ein λ-Operator den Entropie-Strafterm, welcher die Kompressionsrate des neuronalen Netzes bestimmt. Die Effektivität dieser Methode wurde in mehreren Experimenten untersucht. Dazu zählen Bildklassifizierungsaufgaben mit den CIFAR-10, CIFAR-100 und ImageNet Datensätzen und die Kompression bekannter neuronaler Netze (ResNets, EfficientNet) sowie der mittels Modellskalierung erzeugten Netze. Außerdem konnten im direkten Vergleich zum Standard der ternären Quantisierung, Trained Ternary Qantization (TTQ), die Vorteile von EC2T aufgezeigt und neue Maßstäbe gesetzt werden. Zum Beispiel komprimiert EC2T ResNet-20 um mehr als ein 24$\times$ und reduziert die Anzahl arithmetischer Operationen um mehr als ein 12$\times$, bei einem minimalen Genauigkeitsabfall von 0.9%.

This thesis originated in cooperation with the Fraunhofer Institute for Telecommunications, Heinrich Hertz Institute (HHI).

Department of Video Coding & Analytics
Machine Learning Group
Einsteinufer 37
10587 Berlin

Acknowledgments

First of all I would like to thank my supervisor Simon Wiedemann at the Fraunhofer HHI. I have been very grateful to receive his guidance, support and forward-thinking ideas to realize this thesis.

Also, I want to express my gratitude to Professor Klaus-Robert Müller and Professor Thomas Wiegand for supervising me, sharing their expertise and for their exceptional science and teaching at TU Berlin. Thanks to the head of the Machine Learning group, Wojciech Samek, for giving me the opportunity to carry out state of the art research in the field of efficient deep learning.

I want to thank my colleagues for sharing knowledge, giving constructive feedback, helping out and proofreading, above all: Arturo Marbán, Temesgen Mehari, David Neumann, Sören Becker, Maximilian Kohlbrenner, Tamer Ajaj and Dennis Grinwald. Special thanks go to Leander Cascorbi and Samer Al-Magazachi for proofreading.

Finally, I give sincere thanks to my beloved ones and friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the deep learning era, convolutional neural networks (CNNs) have taken the spotlight, due to their outstanding performance in computer vision applications. As CNNs become increasingly deeper and larger, they also become slower and require more computation. Such increases in computational demands make it difficult to execute state-of-the-art CNN models on resource-constrained platforms such as mobile or embedded devices (e.g. smartphones, wearable healthcare devices or Internet of Things). Thus, the design of efficient CNNs has become an active research area in recent years.

In this master thesis a general framework is proposed which generates efficient CNNs for image classification tasks. By joining and extending methods from the research areas of neural network compression and neural architecture search (NAS), this framework yields CNNs that are characterized by significant efficiency improvements in terms of storage requirements and computational complexity.

In this chapter we describe the thesis' objective and the concept of our proposed two-step paradigm to generate efficient CNN representations. Then we point out our motivation and specify the thesis' outline.

1.1 Objective

Given an image classification task, an image dataset and a target classification accuracy, the core objective of this thesis is to

> *find the most efficient neural network representation*
> *which solves the given task to the specified quality level.*

The efficiency of a neural network is largely determined by the number of math operations required to make predictions and the number of bytes required to store the model parameters. Thus, to solve the problem, we employ a two-step paradigm (figure 1.1) which optimizes the computational complexity and the storage footprint of neural networks.

1.2 Method and Contribution

The proposed strategy is part of a general framework which combines methods from the fields of neural architecture search (NAS) and neural network compression. It can be described in two steps:

1. **Creating the Supernet**: by employing *Compound Model Scaling*, a neural network is created which solves the given task with a slightly better accuracy than specified.

2. **Extracting an Optimal Subnet:** simultaneously pruning and quantizing the supernet by applying *Entropy-Constrained Trained Ternarization* (EC2T) yields a storage- and compute-efficient representation of the original supernet.

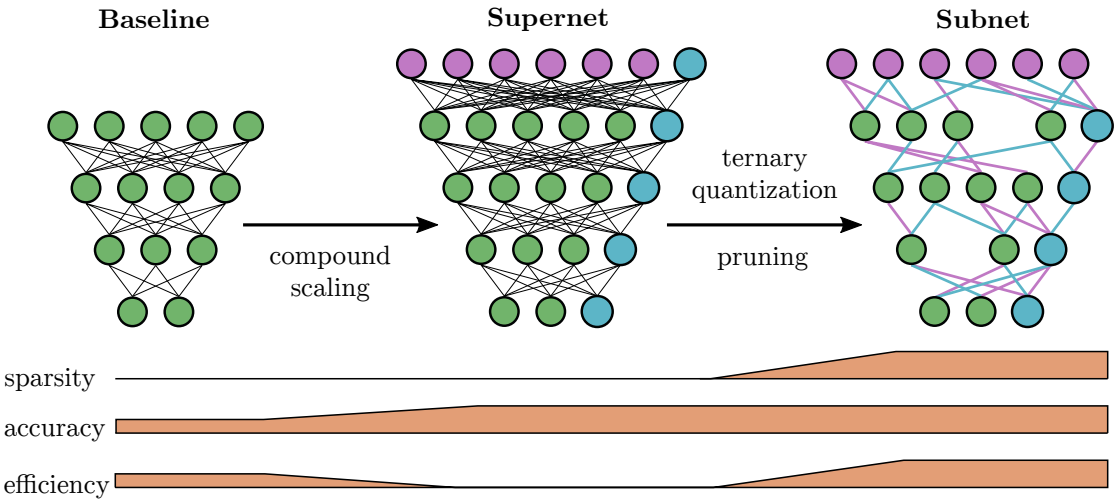

FIGURE 1.1: Firstly, a compound scaling method is applied to a rather small baseline model. Based on a constrained grid search, a well-performing ratio of scaling factors is evaluated, by which the baseline model is extended. The resulting, rather over-parameterized model can be understood as a supernet which includes many redundant connections that do not contribute to the model's performance. The goal is to find a subnet within this supernet which solves the target task to a specified quality level. For this purpose, in a second step, entropy-constrained trained ternarization (EC2T) is applied to the supernet. This algorithm iteratively assigns network connections w_{ij} to only three discrete values (ternary quantization): $w_{ij} \leftarrow \{w_n, 0, w_p\}$. From these values, the so-called centroids, w_n and w_p are learned while training the network (trained ternary quantization). Introducing 0 as a third centroid incorporates network sparsity by deleting connections (pruning).

We employ compound model scaling because it is a NAS method with rather little expense, yet outperforming very time-consuming NAS methods [2]. It has contributed significantly to the current state-of-the-art of efficient CNNs [3]. Furthermore it turned out that randomly sampling architectures from a predefined search space yields good, if not better, results compared to expensive NAS methods based on the same search space [4, 5]. Therefore, it's a legitimate question to ask whether either the search strategy is advantageous in finding innovative architectures, or whether the found architectures owe their success to the search space design.

We chose a ternary quantization process because ternary network representations significantly reduce the computational complexity of convolution operations. As will be shown later on in this thesis, a convolution operation on ternary weight matrices can be implemented through simple additions and two multiplications only. Also, ternary networks have a small storage footprint, since storing a ternary network layer requires only two binary bitmasks plus two discrete centroid values. Additionally, it was shown that ternary neural network execution can be implemented in hardware accelerators very efficiently [6–8].

Sparsifying neural networks is a key method to decrease memory requirements and the number of operations. In order to enhance sparsification, i.e. deleting as many connections as possible while maintaining accuracy, we incorporate an entropy constraint into the assignment function of the quantization process. The constraint penalizes parameters with a low information content and increases the probability of their deletion. Also, incorporating entropy statistics into the objective of the quantizing process predestines the resulting network for subsequent lossless compression techniques like entropy coding or sparse matrix formats (CSR, CER). Having the trained ternary quantization and the entropy constraint, we call our proposed algorithm entropy-constrained trained ternarization (EC2T).

Entropy-constrained trained ternarization (ECT2) is an iterative, dynamic process, enabling what is called "neuroplasticity" in biology. The gradient-based optimization would allow mistakenly pruned connections to get re-assigned to a centroid's cluster. Iterative retraining on the one hand compensates for the deleted connections, and on the other hand strengthens connections which are more established and contributing.

The proposed framework is scalable and applicable to different baseline models. The entropy constraint controls the intensity of network pruning and thus co-determines the compression ratio. With this regulator, the accuracy-efficiency trade-off can be controlled.

1.3 Motivation

A general trend of neural network design has been to find larger models to achieve better performance without considering the memory or power budget. Existing CNNs are computationally expensive and memory intensive, making them prohibitive for devices with low memory resources or in applications with strict latency requirements. Back in 2015, Goodfellow et al. [9] observed that artificial neural networks have doubled in size roughly every 2.4 years. Actually, the growth of neural networks is even faster, considering that some recent architectures consist of more than half a billion parameters [10, 11]. The well-known ResNet-50 [12] with 50 convolutional layers needs over 102 MB memory for storage and over 3.8 billion multiply-accumulate operations (MACs) when processing a single image sample. As the increase in transistor speed due to semiconductor process improvements has slowed down dramatically, it seems unlikely

that mobile processors will meet computational requirements of current neural network architectures [13].

Furthermore, neural networks are typically over-parameterized and there is significant redundancy for deep learning models. Denil et al. [14] demonstrate the redundancies in neural network parameters: they show that the parameters within a neural network layer can be accurately predicted from a small subset of them. This results in a waste of both computation and memory. Eliminating redundancies and designing efficient neural networks is an important step towards enabling the deployment of deep learning in resource constrained systems, such as autonomous vehicles, robots, smartphones, wearables, Internet of Things, etc. In the field of virtual reality, augmented reality, and smart wearable devices, efficient CNNs can enable entirely new on-device experiences and security improvements.

The variety of application domains poses new challenges to the efficient processing of neural networks which also includes adaptability and scalability in order to make deep learning available on a bunch of mobile platforms. Efficient representations of neural networks have a beneficial impact on:

- **Energy Consumption:** Many of the mobile platforms have stringent energy limitations. The energy cost per 32bit operation in a 45nm technology ranges from 0.9pJ for additions to 1300pJ for off-chip memory access [1]. Consequently, the memory access is often the energy-bottleneck for processing neural networks. Each multiply–accumulate operation (MAC), which is the underlying arithmetic of convolutions, requires three memory reads and one memory write [15]. Running large neural networks needs significantly more memory access. As shown in [16], running a large network at only 30Hz requires almost 20W for memory access which is beyond the power budget of a typical mobile device. Thus, efficient processing of neural networks is of prime importance for devices with power constraints and helps preserve battery life. If the model is small enough, it can fit in the on-chip SRAM, which saves energy and is faster to access than off-chip DRAM memory.

- **Latency:** For real-time applications such as self-driving cars or augmented reality glasses, latency is critical to guarantee safety or user experience. Efficient neural networks decrease latency significantly. If the model is small enough this enables "near-sensor processing" and thus further latency caused by cloud computing can be avoided. This is especially interesting for applications which require fast responses and higher levels of data privacy such as wearable healthcare devices [17].

- **Communication Cost:** In many applications, it is desirable to execute neural networks near the sensor. For instance, computer vision applications often generate a huge amount of data. Extracting meaningful information from the video right at the image sensor rather than in the cloud would reduce the communication cost. A smaller model means less overhead when exporting it to clients, e.g. frequent

over-the-air updates for customers' self-driving cars would be more feasible. While currently most of the processing is in the cloud, performing on the device itself would reduce communication cost, latency and dependency on connectivity but also improve security.

- **Privacy:** Neural networks are susceptible to several security vulnerabilities that can be exploited to perform security attacks [18]. Employing efficient on-chip processing of neural networks forms a closed system that is less vulnerable to external attacks. This is desired for applications where the security risks of relying on the cloud are too high.

1.4 Thesis Outline

Chapter 2 provides fundamentals and terminology which is related to the thesis' objective. This includes definitions for convolutional neural networks (CNNs) and their computational cost as well as an introduction to techniques from the field of neural network compression and neural architecture search (NAS) which aim to enhance CNN efficiency.

Chapter 3 reviews related work on efficient neural network design. We focus on work which is related to our approach: search on efficient neural architectures, ternary quantization, trained quantization, model scaling, entropy-constrained quantization and lossless compression.

Chapter 4 describes our proposed framework in detail. We explain our baseline models and picture the two-step paradigm of compound model scaling and entropy-constrained trained ternarization (EC2T).

Chapter 5 presents experimental results and describes aspects of the algorithm's implementation. Also, we validate the proposed approach and compare it with results from related works.

Chapter 6 summarizes the thesis and gives an outlook about future work.

Chapter 2

Fundamentals

In this chapter, we first give an introduction to what is a deep neural network (DNN) and the position of DNNs in the context of artificial intelligence (AI). Then we introduce convolutional neural networks (CNNs) and different types of convolution operations along with their computational cost. In the subsequent sections we describe fundamental techniques and tools from neural architecture search (NAS) and neural network compression. Here we emphasize on topics which are related to our work. For our proposed entropy-constrained quantization technique, we give a short introduction to basic concepts from information theory.

2.1 Deep Neural Networks (DNNs)

Within AI, deep learning is a subfield of machine learning (ML). In contrast to other AI paradigms, *machine learning* algorithms have the ability to acquire knowledge from data, and to improve with experience without being explicitly programmed. They are not restricted to executing logical rules dependent on a hand-crafted knowledge base.

However, extracting high-level features from raw data, e.g. different accents in a speech recognition task, can be challenging. *Deep learning* is a particular kind of machine learning that solves this central problem by learning to represent the world as a nested hierarchy of concepts [9]. By employing (deep) artificial neural networks, deep learning methods enable complex feature representations that are expressed in terms of other, simpler representations. For instance, DNNs can detect complex patterns in image data by combining simpler concepts, such as lines, edges to contours.

Artificial neural networks (ANNs), hereinafter called for simplicity neural networks, consist of neurons and connections. Neurons are organized in layers, but neurons within the same layer are not connected among each other. The first layer of a neural network is called *input layer*, the final layer *output layer* and the layers in between "hidden layers".

Deep neural networks (DNNs) are considered as "deep" when the number of layers is large. Although there is no explicit definition for the minimum number of network layers in order to call it a DNN, modern architectures can consist of hundreds of layers. A single *neuron* within a layer receives numerous inputs from predecessor neurons and generates

one output. The output is a weighted sum of the neuron's inputs to which an *activation function* is applied. Without the activation function, neural network computation would be linear algebra systems. Common activation functions such as $\mathrm{ReLU}(x) = \max(0, x)$, $\tanh(x)$ or $\mathrm{sigmoid}(x) = \frac{1}{1+\exp(-x)}$ introduce nonlinearity which enables the network to approximate any continuous function (universal approximation theorem [19]).

Neurons are wired through *connections*. Each connection transfers the output activation of a certain neuron i to the input of another neuron j. The transferred activation signal can be increased or decreased, dependent on the connection's *weight* w_{ij} which is multiplied with the activation. As illustrated in figure 2.1, the output activation of a neuron j is defined as

$$y_j = a_j \left(\sum_i w_{ij} x_i + b_j \right) \tag{2.1}$$

where w_{ij} are the weights related to the input connections, x_i are the input activation signals, $a_j(\cdot)$ is the activation function of neuron j and b_j its bias term.

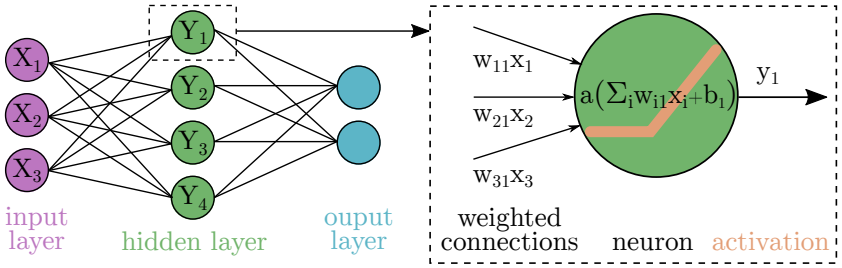

FIGURE 2.1: Three-layer neural network example. The output of a single neuron is the weighted sum of its inputs to which an activation funtion is applied. b_1 is a bias term.

In case of image classification, that is to recognize the input images as belonging to one of a predefined number of classes, the network's output layer usually contains one neuron per class. In order to learn a given task, the network's weight values and biases are adjusted iteratively. This process is called network *training*. Running a (pretrained) network without adjusting the weights, is referred to as *inference* or a feed-forward pass. The goal in learning this task is to determine the weights that maximize the activation of the correct class' output neuron, i.e. its score, and minimize the incorrect output neurons' scores. The difference between the ideal, true scores and the predicted scores can be quantified with error measurements and is called *loss*. To minimize the average loss, *gradient descent* is a popular technique. It is a first-order optimization method by computing the gradient of the loss relative to each weight and updating the weights $\hat{w}_{ij}$ in the negative direction of the gradient such that

$$\hat{w}_{ij} = w_{ij} - \mu \frac{\partial \text{ loss}}{\partial \ w_{ij}} \tag{2.2}$$

where μ is a so-called *learning rate* and the partial derivative of the loss with respect to w_{ij} corresponds to the gradient at w_{ij}.

An efficient way to compute the gradient's partial derivatives is through the *back-propagation* algorithm. To calculate the gradient with back-propagation, firstly each layer's activations are calculated by performing a forward pass. Secondly, the resulting loss is calculated. Next, the gradient of the loss for each weight is calculated iteratively from the output layer to the input layer according to the chain rule. Finally, the weights are updated according to equation (2.2).

The process of feed-forward computation, back-propagation of the loss, and weight update constitutes one iteration of training. Usually, it takes hundreds of thousands of iterations to train a deep neural network [16]. In practice, to speed up and stabilize the training process, the weight update is executed after forward passing multiple samples of input data, i.e. a data *batch*, and backpropagating the average batch loss.

2.1.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are often used in image analysis. Assume the goal is to detect particular objects from a set of image samples of height H and width W. In deep learning, a convolution corresponds to a *kernel* which would move across these input images from top left to bottom right (row-major order). If this kernel consisted of 3×3 trainable kernel weights k_{ij}, the kernel size K would be 3. At each kernel position an element-wise multiplication and addition with the pixel values of the underlying input image i_{hw} would be performed:

$$\begin{matrix} i_{00} & i_{01} & i_{02} & i_{03} & i_{04} & \cdots \\ i_{10} & i_{11} & i_{12} & i_{13} & i_{14} & \cdots \\ i_{20} & i_{21} & i_{22} & i_{23} & i_{24} & \cdots \\ i_{30} & i_{31} & i_{32} & i_{33} & i_{34} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \end{matrix} \quad * \quad \begin{bmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ k_{20} & k_{21} & k_{22} \end{bmatrix}$$

$$\begin{aligned} \textbf{position 0:} \quad m_{00} = {} & i_{00}k_{00} + i_{01}k_{01} + i_{02}k_{02} + i_{10}k_{10} + i_{11}k_{11} + i_{12}k_{12} \\ & + i_{20}k_{20} + i_{21}k_{21} + i_{22}k_{22} \\ \textbf{position 1:} \quad m_{01} = {} & i_{01}k_{00} + i_{02}k_{01} + i_{03}k_{02} + i_{11}k_{10} + i_{12}k_{11} + i_{13}k_{12} \\ & + i_{21}k_{20} + i_{22}k_{21} + i_{23}k_{22} \\ \textbf{position 2:} \quad m_{02} = {} & \cdots \end{aligned}$$

such that one kernel application, i.e. one convolution, requires K^2 multiplications and $K^2 - 1$ additions according to:

$$m_{h,w} = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} i_{h+i,w+j} k_{i,j}. \tag{2.3}$$

Thus, a convolution operation in deep learning basically corresponds to a *multiply–accumulate operation* (MAC). After the kernel was shifted over the input image with a given step size, i.e. *stride*, in both dimensions h and w, an *output feature map* is

generated. It is build by applying an activation function on the convolutional outputs $m_{h,w}$. In a convolutional neural network layer, each output feature map is assigned to a concrete *output channel* of that layer and serves as an input feature map, i.e. input activation, for the subsequent layer, etc.

Usually, a convolution layer has N input channels, i.e. receives many input feature maps. For instance, when dealing with multi-channel data like RGB images, the input layer typically has 3 input channels, one for each color. In this case, an output feature map of a specific output channel is created by a set of 3 kernels. This set of N kernels is referred to as *filter*. A convolution layer with N input channels and M output channels is composed of M filters which are composed of N kernels. Whereas filters are composed of weights, input and output feature maps are composed of activations. The total number of weights within a convolution layer thus amounts to NK^2M. The total number of MACs amounts to $HWNK^2M$, where $H \times W$ indicates the number of activations in the input feature map, i.e. the input resolution.

A convolutional neural network typically consists of several *stages*. A stage is a sequence of building blocks with the same input resolution and layer architecture. Per stage the input resolution is gradually shrunk but the number of channels is usually expanded over layers. For example, ResNets [20] scale the initial input shape ($224 \times 224 \times 3$) to a final output shape of ($7 \times 7 \times 512$), where ($H \times W \times N$). The number of concatenated building blocks, or layers, is referred to as network *depth* and the number of channels as network *width*. In order to downscale the input resolution, *pooling* layers or convolution layers with an increased stride are applied. A pooling operation or pooling filter with a size of 2×2 and a stride of 2 would halve the feature map resolution by either calculating the average or the maximum value of each 2×2 feature map patch (referred to as average or max pooling). Reducing the output feature map size with a convolution layer follows the following schema:

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \cdot \text{padding} - K}{\text{stride}} + 1 \right\rfloor \tag{2.4}$$

where $D_{in} = H = W$ is the input feature map size, D_{out} the output feature map size and K the kernel size, assuming the feature maps and kernels are square. *Padding* controls the amount of implicit zero-paddings on both dimensions of the input feature map. Without padding, each convolution would reduce the feature map resolution, which is not desired.

2.1.2 Computational Cost of CNN Layers

In this section we introduce layer types which are commonly used in CNNs. Also we describe the computational cost of these layers in the form of multiply–accumulate operations (MACs) and floating point operations (FLOPs). Note that, even though activation functions are not listed below, they also require FLOPs. ReLUs for example require HWM multiplications, one per output feature map element. Sigmoids require $2 \times HWM$ multiplications and HWM additions.

Standard Convolution

As depicted in figure 2.2 and described in section 2.1.1, a standard convolution requires $HWNK^2M$ MACs, where H and W indicate the size of the input feature map, N the number of input channels, K the kernel size and M the number of output channels. The apostrophe in figure 2.2 (H') and the following figures represents that weights or activations changed but the dimensionality remains the same.

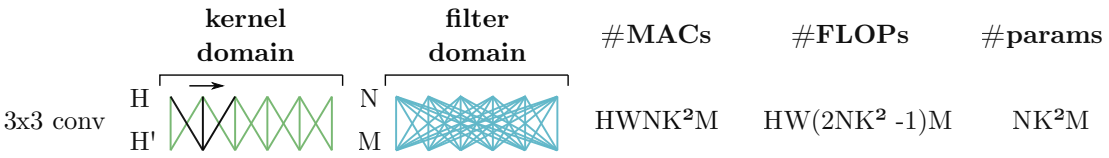

FIGURE 2.2: Kernel and filter domain of a standard convolution with $K = 3$.

According to equation 2.3, the number of MACs refers to $HW(NK^2 - 1)M$ additions and $HWNK^2M$ multiplications. Consequently, the total number of FLOPs for a convolution layer amounts to $HW(2NK^2 - 1)M$ which is almost double the number of MACs. This principle is emphasized because some authors report MACs but mistakenly call it FLOPs.

Pointwise Convolution

A pointwise convolution is a convolution operation with a kernel of size 1×1. It requires $HWNM$ MACs or $HWNM$ multiplications and $HW(N-1)M$ additions, which results in $HW(2N - 1)M$ total FLOPs, respectively. Applying a single 1×1 kernel to an input feature map would correspond to just multiplying a certain number to the input activations.

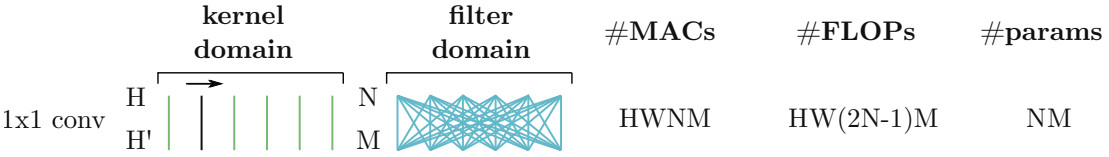

FIGURE 2.3: Kernel and filter domain of a pointwise convolution.

Actually, a pointwise convolution is more than that: by applying a whole filter of 1×1 kernels to N input channels, pointwise convolution can be understand as feature map pooling, see figure 2.3. That is multiplying the HWN input feature map elements with N 1×1 kernels, accumulating the N resulting maps and applying an activation function to it in order to generate one output feature map.

Pointwise convolutions are often used to increase or reduce the number of channels with low computational cost. For the latter purpose, pointwise convolutions are also referred to as "bottleneck" layers. The aim of bottlenecks is to reduce the computational cost of convolutions with larger kernel sizes by firstly reducing the number of channels, then applying the larger kernel on the reduced number of channels and finally expanding the number of channels again with another pointwise convolution. Note, that even

a moderate kernel size of 3×3 or 5×5 can be prohibitively expensive on top of a convolutional layer with a large number of filters.

Grouped Convolution

This special case of convolution layer groups the input feature maps and performs convolution independently for each group. If G denotes the number of groups, each filter is applied to only N/G input feature maps which reduces the total number of operations and network weights. In other words, the output feature maps of a grouped convolution layer are not derived from all input feature maps but only from a fraction N/G of them. In figure 2.4 a grouped convolution with $G = 3$ filter groups is illustrated. The depth of each filter is only one third of that of a standard convolution which results in $HWNK^2M/G$ MACs or $HW(2NK^2 - 1)M/G$ FLOPs respectively.

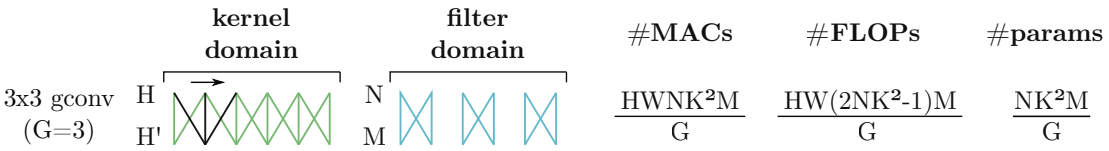

FIGURE 2.4: Kernel and filter domain of a grouped convolution with $G = 3$.

Grouped convolutions are computationally efficient but each filter group only handles information from fixed input feature maps. As information flow between channel groups is restricted, learning a wide variety of features may be limited. To overcome this problem, channel shuffle [21] can be applied to mix information from different filter groups.

Depthwise Convolution

If the numbers of input and output channels are the same and G equals this number, each filter is of depth 1. This special case of grouped convolution is called depthwise convolution. Given there are N input channels, then N filters of size K^2 are applied independently to each input feature map. The computational cost of a depthwise convolution is reduced to $HWNK^2$ MACs which corresponds to $HWN(2K^2 - 1)$ FLOPs.

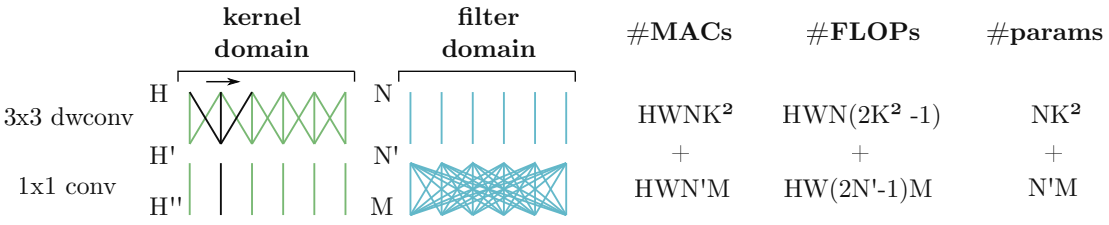

FIGURE 2.5: Kernel and filter domain of a depthwise separable convolution with $K = 3$.

Depthwise convolutions are often used in combination with pointwise convolutions to decompose a standard convolution. As depicted in figure 2.5, the computational

complexity of depthwise plus pointwise convolution sums to $HWNK^2 + HWNM = HWN(K^2 + M)$ MACs which is more efficient than $HWNK^2M$ MACs required for a standard convolution. This decomposition scheme is referred to as *depthwise separable convolution*.

Residual Connections

Residual connections, also referred to as skip connections, shortcut connections or identity connections, map the input activations of a certain layer to its output or to the output of another, more deeply located layer. This technique can be understand as propagating information in an additional parallel path to deeper structures which is certainly helpful in very deep networks to avoid the vanishing gradient problem.

In the simplest case, when just adding the identity, the computational cost amounts to HWM additions. But there also exist skip connections with trainable skip weights, i.e. when skipping to a layer with more input channels, skip weights could be implemented by a pointwise convolution which expands the channel dimensionality accordingly.

BatchNorm Layers and Global Averaging

To increase the stability of a neural network, batch normalization normalizes feature maps while network training. This is useful because the distribution of activations is constantly changing. Assuming that some activations are very high compared to others, this can slow down the training process because each layer must learn to adapt to a new distribution in every training step. A BatchNorm layer learns mean and variance statistics and normalizes the activations after a batch of data samples has passed the network. When the model is trained, mean and variance parameters from the BatchNorm layer can be merged as a bias term which is added on the previous layer's output. Such that BatchNorm layers count M parameters and require HWM addition operations.

Global average pooling is an operation that calculates the average output of each input feature map. To calculate the average, global average pooling requires $N(HW-1)$ additions and N divisions. It is commonly used in the final stage of a CNN to squeeze out one resulting value from the output feature maps which is then used to calculate a class score in a classifaction task. In many cases global average pooling is followed by one or more fully connected layers before calculating the score.

Fully Connected Layer

As mentioned above, in classification tasks fully connected layers (FC) are used in CNNs to take the flattened or averaged results of the convolution layers and use them to learn the final classification. For instance, if the final convolution layer has 1000 output feature maps they could be averaged to 1000 discrete values. These values serve as input for the FC layer. If 100 classes exist, the FC layer maps $N = 1000$ input activations to $M = 100$ output activations which represent probabilies that certain features belong to a certain

class label. A fully connected layer requires NM parameters, NM multiplications and $(N-1)M$ additions.

Convolution layers are not fully connected because, by employing parameter sharing, they require only K^2 weights to map HW input activations to HW output activations (assuming appropriate padding and one channel).

Squeeze-and-Excitation Module

Squeeze-and-Excitation (SE) modules [22] improve channel interdependencies at almost no computational cost. Basically, the idea is to add parameters to each output channel so that the neural network can adjust the weighting of each feature map. Such module is illustrated in figure 4.2 later on in this thesis.

The first operation of the SE module is global average pooling which squeezes out one value per output channel. These N values serve as input for a subsequent fully connected layer (FC1). The number of output activations is typically reduced by a ratio r such that FC1 requires NN/r MACs. The second fully connected layer FC2 expands the output dimension back to the original level (N/rN MACs) and is followed by a sigmoid activation. The resulting values are multiplied with the N input feature maps to weight them. This step requires HWN multiplications.

The SE module usually improves network accuracies and can be added to any model. As, due to pooling, the input feature maps are squeezed to 1×1, the fully connected layers are often implemented as pointwise convolutions, i.e. $HWNK^2M$ is reduced to NM as $H = W = K = 1$.

2.2 Neural Architecture Search (NAS)

Designing deep neural network architectures has been an active research area in recent years. Both manual and automated search strategies have played an important role in facing the trade-off between accuracy and efficiency metrics. The common practice of NAS is to first define a search space and then finding optimal architectures in this search space by employing prior knowledge, human expertise and systematic or automated search strategies.

Currently, the automation of neural architecture search is one of the fastest developing areas in machine learning. Key components of a successful automated NAS are the search space design and the choice of an appropriate search strategy. The search space is often exponentially large: by assuming that the design space offers only three different kernel sizes $\{1, 3, 5\}$ and five different filter numbers $\{32, 64, 128, 256, 512\}$, for each of the 16 layers of an exemplary CNN, the number of possible architectures amounts to $(3 \times 5)^{16} \approx 6 \times 10^{18}$. Automated neural architecture search guides the exploration of a given search space by employing a search strategy.

Search Spaces for neural networks can be classified into a global and a cell-based category. The former is defined for graphs that represent an entire neural architecture

and the latter focuses on repeated small graphs which built a network. Hand-crafted architectures often consist of a repetition of fixed structures which are referred to as *cells* or *blocks*. In the cell-based search space only the cells' hyperparameters vary while the topology of all cells remains the same. This kind of search space typically has a reduced size compared to global search spaces. Still, employing cell-based architectures requires a macro-architecture that specifies how to stack the cells. Search spaces are parameterized by the number of nodes, or number of layers respectively, the respective layer operation, e.g. pooling, convolution or more advanced (sequences of) operations, hyperparamters associated with these operations, e.g. number of filters, kernel sizes or strides and, optionally, constraints which could be based on prior knowledge. This includes, for instance, avoiding architectures which are known to perform poor or which are inefficient. Constraints minimize the search space which needs to be explored. However, by constraining the search space, and thus incorporating prior knowledge, human bias could be introduced. This may prevent to find novel architectures that go beyond the current human knowledge.

Search Strategies used to explore the defined search spaces are mostly based on reinforcement learning (RL) [2, 3, 23–27], differentiable neural architecture search (DNAS) [28–32] and evolutionary algorithms (EA) [33, 34] but also random search [35, 36], model scaling [27], sequential model-based optimization (SMBO) [37], Bayesian optimization [38] and Monte Carlo Tree Search [39] are applied.

Model Scaling

In model scaling a baseline neural network is scaled up or down by changing the depth or width of the network, or the size of the input image. It was common to scale only one of this three dimensions until EfficientNets [27] achieved outstanding benchmarks on several image classification tasks with their compound model scaling approach.

Compound scaling scales network width, depth and resolution with a set of fixed scaling coefficients. Tan and Le [27] observe that scaling up any dimension improves accuracy, but the accuracy gain diminishes for bigger models. Another observation is that different scaling dimensions are not independent: for higher resolution images, network depth should be increased, such that the larger receptive fields can help capture similar features that include more pixels in bigger images. Correspondingly, also network width should be increased for higher resolution images, in order to capture more fine-grained patterns with more pixels [27].

Model scaling simplifies the search problem for resource constraints, but it still remains a large design space to explore different depths, width and resolutions for each network stage. The effectiveness of model scaling heavily depends on the baseline network.

2.3 Neural Network Compression

In general, there exist two ways to compress neural networks, namely: (a) reducing the number of operations and model size; and (b) decreasing the precision of operations and operands. Techniques pursuing these approaches can be categorized into three groups:

- **Parameter Reduction**: getting rid of redundant and unimportant model parameters. These techniques can be further classified into pruning, structured matrix, low-rank decomposition, transferred convolution filters and knowledge distillation.
- **Parameter Quantization**: reducing the precision of model parameters and thus the absolute number of bits to represent them. Quantization can be further classified to uniform quantization, nonuniform quantization, entropy-constrained quantization, binarization and ternarization.
- **Lossless Compression**: exploits sparsity and low-entropy statistics of already pruned and/or quantized model parameters to further compress them. These techniques can be further classified into compressed matrix formats and entropy coding.

Other approaches to reduce the convolutional overheads include using FFT based convolutions and fast convolution using the Winograd algorithm [40]. Existing deep learning platform libraries, such as MKL and cuDNN, dynamically choose the appropriate algorithm for given filter shapes and sizes [15]. Due to the low-level implementation of this kind of approaches it is not common to manipulate them.

Neural Network Pruning

As DNNs are usually overparameterized, a large number of network parameters is redundant and can be removed. Zeroing out parameters, and thus creating sparse network matrices, is called pruning. The idea is to get rid of the most unimportant weights, i.e. weights which have the least impact on the neural network's output. The weights' amplitudes are a commonly used metric to delete parameters beneath a certain threshold.

Other importance metrics are based on weight norms, the eigenvalues of the correlation matrix found by Principal Components Analysis (PCA), similarity measurements based on the Euclidean distance [41], second-derivative information [42], Taylor expansion [43] or the scaling factors of the Batch Normalization layers [44].

In the field of neural network pruning one can distinguish between element-wise (intra-kernel) and structured pruning. Whereas element-wise pruning aims to prune single network connections, structured pruning tackles pruning of kernels, filters, channels or whole layers. Element-wise pruning and kernel pruning result in sparse kernel matrices, while filter, channel or layer pruning reduce the width and depth of the neural network's macro architecture respectively. In many approaches, subsequently to pruning, a retraining process adjusts the remaining weights and thus compensates for the deleted connections.

Neural Network Quantization

Parameter quantization reduces the number of bits required to represent each weight of the original network and thus can be seen as a precision reduction. Quantization maps data to a smaller set of quantization levels. The objective is to minimize the error between the reconstructed quantized data and the original data. The more quantization levels are provided, the higher is the resulting precision. Having n quantization levels, the number of bits required to represent a data point is $\log_2 n$, i.e. quantizing thousands of discrete weight values of a neural network's hidden layer to 256 specified quantization levels n makes each of the thousands of weights representable in 8bit. The default baseline precision used on GPU and CPU platforms is single-precision floating-point (occupying 32bits) which comes with a high computational cost, power consumption and arithmetic operation latency [45, 15].

The most intuitive way to specify quantization levels is by mapping the data to levels of uniform distance. To exploit the nonlinear distribution of weights within a layer, other quantization schemes include nununiform mapping functions, e.g. k-means clustering which is determined by the weight's distribution, see figure 2.6.

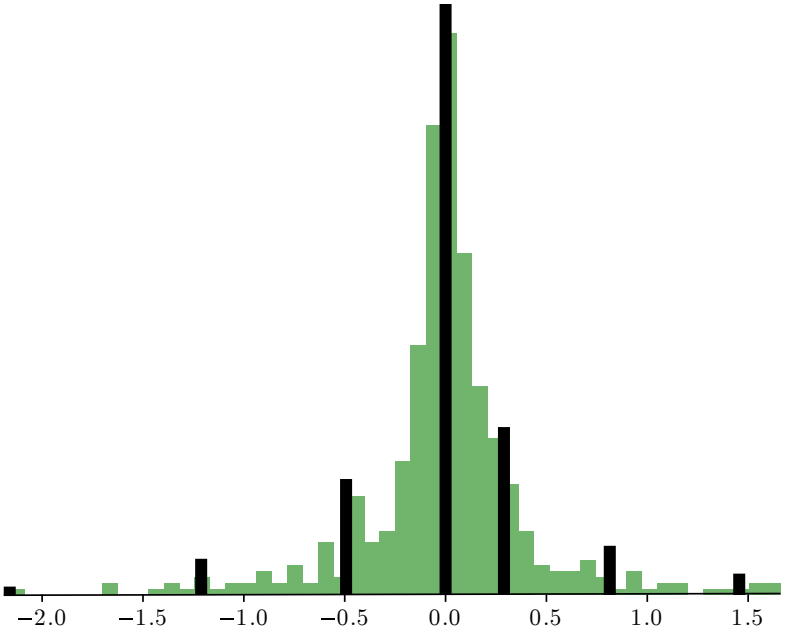

FIGURE 2.6: Quantizing a neural network's layer weights to 7 discrete quantization levels which are also called centroids. The centroids (black bars) were generated by k-means clustering and the height of each bar represents the number of layer weights which is assigned to the respective centroid.

Reducing the precision more aggressively to a single bit is referred to as binarization. In this case, weights are assigned to either -1 or $+1$ which allows to replace many multiply-accumulate operations (MAC) by simple additions. However, this kind of aggressive quantization results in great drops in accuracy.

In effort to improve the resulting accuracy drop, the binary weights can be scaled to $-w$ and $+w$. Retaining the first and last layer of the network at full precision and

performing normalization before convolution can also improve the accuracy drop. By allowing weights to be zero, i.e. constraining them to either $-w$, 0 or $+w$, extends the binary approach to a ternary weight network which enables network sparsity. While low-precision networks lead to significant network compression, they often require higher numbers of neurons to achieve accuracies comparable to their floating-point counterparts [45].

It should be noted that the precision of the internal values of a fixed-point MAC operation is typically higher than the weights' and activations' precision. To guarantee no precision loss, weights and input activations of b bit fixed-point precision would require an b bit $\times$ b bit multiplication which generates a $2b$ bit output product. That output would need to be accumulated with $2b + f$ bit precision, where f is determined by the largest filter size: $f = \log_2(N \times K^2)$ with N being the number of input channels and K the kernel size. After accumulation, the precision of the final output activation is typically reduced to b bit [15].

2.4 Lossless Neural Network Compression

By exploiting the sparsity and low-entropy statistics of already pruned and/or quantized weight matrices, lossless compression aims to maximally compress those matrices in a lossless manner. That is, the original matrix format is exactly reconstructable from the compressed format. These techniques can be categorized into compressed matrix formats and entropy coders. Whereas the latter require decoding for network inference, compressed matrix formats can be used to execute MACs efficiently without any decoding steps.

Information Theory

In information theory, entropy coding, also referred to as lossless coding, describes coding algorithms that allow the exact reconstruction of the original source data from the compressed data representation. Lossless coding can compress the original data, if the data contains dependencies or statistical properties that can be exploited [46].

Assuming data is generated by a probabilistic source, information theory provides concepts to describe the minimum information contained in this data. In his landmark paper [47], Claude Shannon states that the minimum information required to fully represent a data element w that has probability $P(w)$ is of

$$I = -\log_2 P(w) \text{ bit.} \tag{2.5}$$

I is referred to as information content. Data elements which occur frequently, i.e. have a high probability, have a low information content, and vice versa for infrequently occuring

elements. Another key measure in information theory is entropy. It is defined as

$$H_P(\mathcal{W}) = -\sum_{w \in \mathcal{W}} P(w) \log_2 P(w) \tag{2.6}$$

and denotes the minimum average number of bits required to represent any element $w \in \mathcal{W} \subset \mathbb{R}^n$. In other words, entropy is the theoretical limit of the average bit-length needed in order to compress data in a lossless manner. The entropy H of a quantized neural network can be defined as

$$H_P(\mathcal{C}) = -\sum_{c \in \mathcal{C}} P(c) \log_2 P(c) \tag{2.7}$$

where $\mathcal{C}$ is a set of quantization levels, i.e. centroids, and $P(c) = |c|/n$ is the ratio of the number of weights assigned to centroid c to the number of all network weights n.

Entropy Coders

According to equation (2.5), entropy coding compresses binary input elements w to output codewords of a length which is approximately proportional to $-\log_2 P(w)$ bits. Thus, more frequently appearing elements are represented by fewer bits, and less frequently appearing elements are represented using more bits. This variable-length coding scheme can be used to further compress a already quantized neural network.

Huffman code is such a variable-length entropy coding strategy. Experiments in [16] show that Huffman coding can save 20% to 50% of neural network storage. However, in practice Huffman coding can require large codeword tables, be computationally complex and produce a bitstream with more redundancies than principally needed [46]. As an alternative entropy coding strategy, *arithmetic coding* can be considered. It does not require the storage of a codeword table as the arithmetic code for a sequence of input elements w is iteratively constructed. It is well-known that arithmetic coding outperforms other coding strategies with regards to both, compression rate and coding efficiency [48, 46].

CABAC is a context-adaptive binary arithmetic coder which is adaptable to DNNs [48]: Firstly, *DeepCABAC* binarizes the input data to bins. The bins are constructed by applying a series of binary decisions on input data, e.g. whether or not the input element is zero or if it has a positive or negative sign. After binarization, a binary probability model is assigned to each bin. Finally, an arithmetic coder efficiently and optimally codes each bin, based on the respective probability model. The probability models, which are also named context models, are updated during the coding process dependent on the local data statistics. This makes CABAC adaptive to different data distributions.

Compressed Matrix Formats

When DNN compression permanently deletes certain weights, i.e. sets them to zero, one challenge is to determine how to store sparse weight matrices in a compressed format. Compression can be applied either in row or column order.

The compressed sparse row (CSR) format can be used to perform sparse matrix–vector multiplication. It scans the non-zero matrix elements in row-major order and stores them in an array V. Simultaneously, it stores the associated column indices in a second array C. In a third array R, pointers are stored which indicate when a new row starts. A 75% sparse $m \times n$ matrix

$$\mathbf{M} = \begin{pmatrix} 0 & 2 & 0 & 4 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

can then be represented by $V = [2, 4, 2, 2, 4, 2]$, $C = [1, 3, 0, 2, 0, 5]$ and $R = [0, 2, 4, 5, 6]$, requiring $2a + m + 1$ numbers, where a is the number of non-zero elements and m the number of rows.

When quantized weight matrices have low entropy statistics, the compressed entropy row (CER) format is provably more optimal than CSR with regards to both, compression and execution efficiency [49]. For the CER format, only the unique values of $\mathbf{M}$ are stored in an frequency-major order, i.e. $V = [0, 2, 4]$. The format is especially beneficial to matrices that consists of many elements which share the same value. Furthermore, column indices for the most frequent value in V, i.e. $V[0] = 0$, are not necessary to reconstruct $\mathbf{M}$. A supplementary pointer array indicates the next unique element's index. The introduced formats not only reduce memory consumption but can also reduce the number of operations and time complexity when performing dot products.

Chapter 3

Related Work

A comprehensive survey on neural architecture search (NAS) was conducted by Elsken et al. [4] and Wistuba et al. [5]. The field of neural network compression and efficient hardware for DNNs has been surveyed by Sze et al. [15] and Wang et al. [45]. Multi-objective NAS is closely related to network compression: both aim at finding well-performing but efficient architectures. Hence, some compression methods can also be seen as NAS methods and vice versa.

3.1 Neural Architecture Search (NAS)

Carefully balancing accuracy and resource-efficiency results in a potentially huge design space. Hand-crafting efficient neural networks therefore requires substantial human efforts. The number of parameters and mathematical operations is reduced by using lower-cost convolutions and reduced filter sizes, like in SqueezeNet, ShuffleNet and their successors [13, 21, 50, 51]. MobileNet V1 [52] introduced depthwise separable convolutions as an efficient replacement for traditional convolution layers. MobileNet V2 [53] further improved performance and efficiency by utilizing resource-efficient inverted residual blocks with linear bottlenecks. These MobileNet building blocks revolutionized the field of mobile CNNs and thus often serve as a backbone or baseline structure for other NAS approaches [2, 3, 27–29].

Early automated NAS methods searched for high-accuracy neural network architectures without considering resource limitations on mobile devices. Therefore, their found architectures (e.g., NASNet [25] , AmoebaNet [33]) are computationally expensive.

More recent, hardware-aware multi-objective problem formulations embed measures of resource efficiency into their main objectives. Incorporating the hardware latency results in specialized and more efficient neural networks for custom hardware platforms [3, 28, 29]. Chu et al. [34] produce mobile GPU-aware networks with an evolutionary algorithm (EA) based search strategy that considers model accuracy, latency and the number of parameters as objectives. Howard et al. [2] blend RL-based NAS with novel architecture advances and NetAdapt [54] to build the third generation of MobileNets. NetAdapt is an automated algorithm to adapt a pretrained network to a mobile platform.

In [27], the MnasNet search space is utilized to find the so-called EfficientNet-B0 baseline model. Unlike Tan et al. [3], here the number of FLOPs are optimized rather than latency. Having found a good baseline model, they apply model scaling to it. The resulting EfficientNets (B1-B7) and their most recent modifications (EfficientNet-L2 [11]) mark the current state-of-the-art and achieve the highest ImageNet top-1 accuracies to date, while being smaller and faster than other CNNs.

3.2 Neural Network Compression

In the field of network pruning, the weights' amplitudes are a commonly used importance metric. Other approaches propose Taylor expansion [43], QR factorization [55] or the scaling factors of the Batch Normalization layers [44] as criterion for kernel and channel importance. Riera et al. [56] compare magnitude-based pruning with importance metrics based on weight norms, the eigenvalues of the correlation matrix, similarity measurements as defined in [41] and random pruning. They state that importance heuristics for pruning are irrelevant and that only the amount of pruned connections is important since retraining adjusts the remaining weights.

Whereas many approaches force element-wise (intra-kernel) pruning, in [57–60] work on structured pruning along channels, filters and layers is investigated. In most approaches connections are pruned irreversibly. Therefore, Dynamic Network Surgery [61] aims to re-establish mistakenly pruned connections. Besides pruning, other parameter reduction techniques include low-rank decomposition [62–64], knowledge distillation [65, 66], structured matrices [67, 68] and transferred convolutional filters [69, 70].

When it comes to model quantization, experiments in [71] show that DNNs can be trained using 16 bit fixed-point representation with little to no degradation in accuracy. Marchisio et al. [18] obtained that 12 bit fixed-point quantization of weights and activations is the optimal design choice for computing inference. Further reduction can be obtained, but it may require retraining to compensate for the accuracy loss. While most research focuses on reducing the bitwidth for inference, Zhou et al. [72] tackle quantizing weights, gradients and activations to accelerate the backward pass as well.

Reducing the precision more aggressively to a single bit was introduced by BinaryConnect [73], where weights are constrained to either $+1$ or -1. By binarizing weights and activations Courbariaux et al. [74] demonstrate that a MAC can be replaced by a 1bit XNOR-count operation. However, this kind of aggressive quantization results in great drops in accuracy. In effort to improve the resulting accuracy drop, Rastegari et al. [75] introduce scaling factors for the binary weights, i.e. constrain them to either $+w$ or $-w$. Ternary weight networks (TWNs) [76] then allowed weights to be zero, i.e. constraining them to either $-w$, 0 or $+w$ which yielded results that outperformed the binary counterparts.

In Trained Ternary Quantization (TTQ) [77], parameters are represented in the form $\{w_l^-, 0, w_l^+\}$, wherein w_l^- and w_l^+ are layerwise trainable parameters with l being the layer index. TTQ back-propagates the loss of the ternary network to the scaling factors

w_l^- and w_l^+ and to a background model which is a full precision copy of the original network. The former enables learning the scaling factors and the latter enables learning the ternary assignment.

Generating efficient neural network representations can also be a result of combining multiple techniques. Deep compression [78] is a three-stage model compression pipeline. First, redundant connections are pruned iteratively. Next, the remaining weights are quantized. Finally, entropy coding is applied to further compress the weight matrices in a lossless manner. Whereas Deep compression splices pruning and quantization, CLIP-Q is a method which performs both steps in parallel [79].

Compressing neural networks under an entropy-constrained optimization was considered by Choi et al. [80] and Wiedemann et al. [81]. By exploiting the sparsity and low-entropy statistics of already pruned and/or quantized weight matrices, lossless compression techniques such as sparse matrix representations [49] and entropy coding [48, 78] are proposed to reach further compression gains.

Chapter 4

Compound Scaling & Entropy-Constrained Trained Ternarization

Our proposed framework merges methods from neural architecure search (NAS) and neural network compression in order to generate efficient CNNs. In short, the approach includes compound scaling of a baseline model plus a subsequent quantization and pruning process. By scaling up the model's width and depth, an overparameterized network is created which is just wide and deep enough that a given task can be solved to a specified quality level.

Entropy-constrained trained ternarization (EC2T) then compresses the neural network significantly, achieving that whole channels can be deleted and the resulting kernel matrices are not only very sparse but are also characterized by a range of just two discrete values per layer: the centroids. By incorporating an entropy constraint into the cost function of the ternary assignment, a heuristic based on the information content of the centroids is introduced in the ternarization process. The constraint enhances network sparsification and thus enables simultaneously pruning and quantizing the network during the gradient-based training of the centroid values.

4.1 Compound Model Scaling

Based on their observation that carefully balancing network depth, width and resolution can significantly improve the accuracy of a model, Tan and Le [27] proposed a new model scaling method. It uniformly scales all three dimensions (depth/width/resolution) using a simple yet highly effective compound coefficient. We adopt this approach in order to find networks which solve given image classification tasks to a specified quality level.

We employ different baseline models for different image classification tasks. In the following section those baseline architectures are introduced. Afterwards we explain the concept of model scaling which is done in two steps: first, a grid search on appropriate

model-multipliers is executed, and second, the best performing multipliers are upscaled uniformly.

4.1.1 Baseline Models

Baseline Model for the CIFAR Database

In order to solve the CIFAR classification tasks we use a rather simple cell micro architecture to create the baseline model. Fragmented and complicated cells are not hardware friendly, and the actual efficiency on mobile devices is low [28]. Han et al. [82] evaluated different ResNet [20] building blocks by altering the stacked elements inside. The stack shown in figure 4.1 yields the best performance and hence serves as the micro structure we use for the CIFAR baseline model.

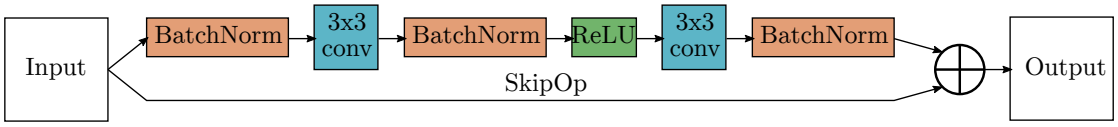

FIGURE 4.1: Building block for CIFAR baseline model

As a macro structure, the ResNet-44 architecture is utilized and adjusted for CIFAR as prescribed in the original paper [20]. It has three stages, containing seven building blocks (figure 4.1) each. The number of filters increases from 16 in the first stage to 64 in stage three. There is one "stem convolution" above the stages which is the input layer and remains unquantized. The network "head" is formed by global average pooling and a fully connected layer. In the following sections we will refer to the CIFAR models as *MicroNet-C10* and *MicroNet-C100*, solving CIFAR-10 and CIFAR-100, respecitvely. The full model is described in table 4.1.

TABLE 4.1: Baseline model for CIFAR tasks. d is a multiplier for scaling depth and w for scaling width. For the baseline $d = w = 1$. #classes is 10 for CIFAR-10 and 100 for CIFAR-100.

Stage	Operation	Resolution	Out Channels	Repetitions
	"stem conv" (3×3) + BN & ReLU	32×32	$16 \times w$	1
1	CIFAR building block	32×32	$16 \times w$	$7 \times d$
2	CIFAR building block	16×16	$32 \times w$	$7 \times d$
3	CIFAR building block	8×8	$64 \times w$	$7 \times d$
	ReLU & Pooling	8×8	$64 \times w$	1
	Fully Connected	1×1	#classes	1

Baseline Model for the ImageNet Database

For the ImageNet task we use EfficientNet-B0 [27] as a baseline model. Compared to other models achieving similar ImageNet accuracy, EfficientNets are much smaller

by exploiting efficient convolution types such as depthwise convolutions and pointwise convolutions. That is why we chose EfficientNet as a baseline to further minimize FLOPs and storage.

The main building block is a mobile inverted bottleneck (MBConv block) as introduced by Sandler et al. [53]. Furthermore it is extended with a squeeze-and-excitation (SE) module [22] and swish activation functions, where $\text{swish}(x) = x \cdot \text{sigmoid}(x)$. The SE-extended MBconv block is depicted in figure 4.2.

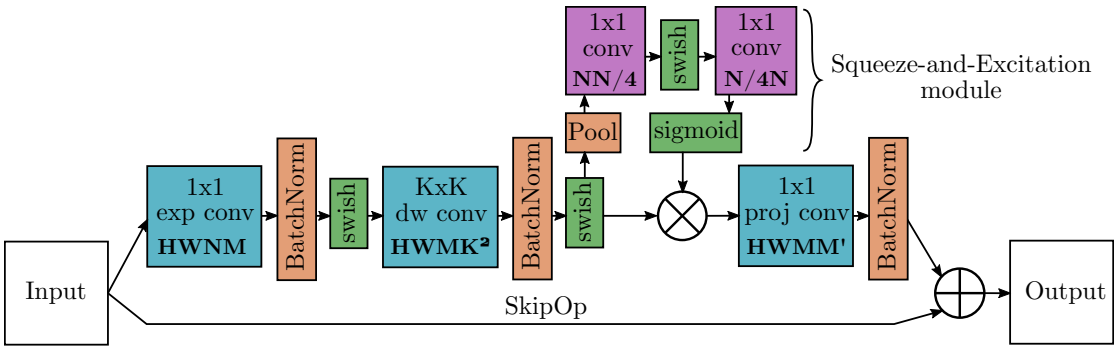

FIGURE 4.2: Building block for the ImageNet baseline model. The computational cost of the convolution operations is highlighted in bold. H and W denote height and width of the input image, N is the number of input channels of the "exp conv", M the number of output channels of the "exp conv" and K the kernel size of the depthwise convolution ("dw conv").

The macro structure of EfficientNet-B0 was found by employing the same search space and search strategy as MnasNet [3] which is based on a reinforcement learning based neural architecture search (NAS). The only difference is that, instead of latency, Tan and Le [27] optimized the number of FLOPs to create EfficientNet-B0. The NAS on different sets of kernel sizes, layer repetitions and filter sizes revealed the network described in table 4.2 which we use as ImageNet baseline model.

4.1.2 Grid Search on Scaling Factors

To scale the baseline models in depth and width one factor is multiplied with the number of building blocks per stage and one with the number of input and output channels per layer. The product of depth multiplier d and number of building blocks is rounded by the ceiling function. The product of the width multiplier w and the number of channels is also rounded by a ceiling function but such that the result is a multiple of 8.
In order to find the best performing ratio of scaling coefficients d, w and r, a grid search

$$\begin{aligned} \text{s.t.} \quad & d \cdot w^2 \cdot r^2 \approx 2 \\ & d \geq 1, w \geq 1, r \geq 1 \end{aligned} \tag{4.1}$$

is applied. The width factor w and resolution factor r are squared because doubling the network width or input image resolution increases the number of FLOPs by four times.

TABLE 4.2: Baseline model for the ImageNet task (EfficientNet-B0). K is the kernel size of the depthwise convolutions, E is the expansion ratio of input channels to output channels of the "exp conv" inside the MBConv blocks (i.e. M=E×N, cf. figure 4.2), d is a multiplier for scaling depth and w for scaling width. For the baseline $d = w = 1$.

Stage	**Operation**	**Resolution**	**Out Channels**	**Repetitions**
	"stem conv" (3×3) + BN & swish	224×224	$32 \times w$	1
1	MBconv, E=1, K=3	112×112	$16 \times w$	$1 \times d$
2	MBconv, E=6, K=3	56×56	$24 \times w$	$2 \times d$
3	MBconv, E=6, K=5	28×28	$40 \times w$	$2 \times d$
4	MBconv, E=6, K=3	14×14	$80 \times w$	$3 \times d$
5	MBconv, E=6, K=5	14×14	$112 \times w$	$3 \times d$
6	MBconv, E=6, K=5	7×7	$192 \times w$	$4 \times d$
7	MBconv, E=6, K=3	7×7	$320 \times w$	$1 \times d$
	"head conv" (1×1) + BN & swish	7×7	$1280 \times w$	1
	Pooling	7×7	$1280 \times w$	
	Fully Connected	1×1	1000	1

Like in [27] the overall scaling amount is constrained to a factor of 2, which ensures that the number of FLOPs will be increased by approximately 2.

For ImageNet we use the reported EfficientNet scaling factors from [27]. The input image resolution for the CIFAR classification tasks is not scaled because the resolution is the same for all samples: 32×32 pixels, respectively. Hence the grid search is performed using a grid of six tuples of (depth, width)-multipliers, satisfying the constraints in 4.1 with $r = 1$:

$$(1.0, 1.4),\ (1.2, 1.3),\ (1.4, 1.2),\ (1.6, 1.1),\ (1.7, 1.1),\ (1.9, 1.0). \tag{4.2}$$

This simplifies the search space, compared to other architecture search methods where the search takes several days on multiple TPU devices [3, 5]. As described in chapter 2.2, model scaling can outperform complex NAS processes and has contributed significantly to the state-of-the-art performance on ImageNet utilizing EfficientNets.

4.1.3 Upscaling the Model Uniformly

In a subsequent step to the grid search the baseline model is upscaled exponentially, according to

$$\hat{d}^{\phi},\ \hat{w}^{\phi},\ \hat{r}^{\phi} \tag{4.3}$$

with $\hat{d}$, $\hat{w}$ and $\hat{r}$ being the best performing scaling factors s.t. 4.1. For any ϕ the total number of FLOPs will increase by approximately 2^{ϕ}.

4.2 Entropy-Constrained Trained Quantization

By applying compound model scaling we find a supernet which solves a given task with a slightly higher accuracy than specified. In a second step, we aim to find an efficient representation of this supernet by applying entropy-constrained trained quantization.

Our proposed method simultaneously quantizes and deletes model parameters in order to extract an efficient subnet out of the initial supernet. The quantization points, which we call centroids, are learned by gradient-based optimization, and therefore the approach is called a "trained quantization". One out of c centroids w_c is forced to take the value zero ($w_0 = 0$). The other centroids are updated based on the cross-entropy loss of the quantized network.

Inspired by the work of Zhu et al. [77], we use the gradients of the quantized network not only to update the centroids w_c but also to update a full precision background model which is a copy of the initial supernet. After having updated the background model and the centroids, the parameters of the background model are assigned to c clusters. Weights within a cluster share the same centroid value. By assigning the parameters of the background model to the clusters, an assignment mask is created which is used to build the quantized network.

The heuristic we use to assign model weights $\mathbf{W}$ to a cluster is based on the squared distances of weight values to centroids and the information content of the clusters. We define the assignment cost function C of layer l as follows:

$$C_c^{(l)} = d(\mathbf{W}^{(l)}, w_c^{(l)})^2 - \lambda^{(l)} \cdot \log_2 P_c^{(l)} \tag{4.4}$$

where c is the cluster index, w_c the centroid value of cluster c, $\mathbf{W}$ the weight matrix of the given layer, $d(\cdot)^2$ the squared distance function and λ a scaling factor for the entropy constraint. $P_c^{(l)} = \#w_c^{(l)} \ / \ \#\mathbf{W}^{(l)}$ is the ratio of number of weights per cluster to the number of all layer weights.

In the case that $\lambda = 0$, the decision boundary between two neighboring clusters would be equidistant to the cluster centroids, i.e. the weights would be assigned in a nearest neighbor manner. When applying the entropy constraint, i.e. $\lambda > 0$, this boundary shifts towards the centroid whose cluster has a higher information content (see figure 4.3). The amount of that shift depends on the effective information content of the actual cluster. If the information content of a cluster c is comparatively high, the shift of the decision boundary into the direction of the according centroid w_c is comparatively small. Consequently, by penalizing the assignment cost of high-information clusters, we ensure that this information is maintained and maximized while simultaneously accumulating weights in the low-information clusters.

In information theory, entropy is the theoretical limit of the average bit-length needed to compress data in a lossless manner. Therefore, we aim to minimize the overall entropy of our neural network. This is done by a) minimizing the number of clusters and b) accumulating weights in one certain cluster to minimize its information content. The latter implies that only a few clusters with high information content remain. Having

one cluster with a very low information content and only a few clusters with very high information content minimizes the overall network entropy.

Usually, a great amount of weight values is close to zero and thus w_0 is the cluster to which the most values are assigned to, i.e. w_0 is usually the cluster with the lowest information content. Incorporating the entropy constraint into the assignment function hence boosts network sparsity as many weights will be assigned to w_0 and thus will be deleted. Training the centroids during the quantization process enables a high degree of adaptivity which compensates for the deletion of model parameters and increases model capacity compared to static quantization techniques or symmetric assignment functions.

4.2.1 Special Case: Entropy-Constrained Trained Ternarization (EC2T)

Restricting the number of clusters to $c = 3$ is a special case of entropy-constrained trained quantization where network parameters are either assigned to the negative valued centroid w_n, to $w_0 = 0$ or to the positive valued centroid w_p. Ternary networks have a small storage footprint, because storing a ternary network layer requires only two binary bitmasks plus two discrete centroid values. For instance, when having a convolutional layer with $N = 32$ input channels, $M = 64$ output channels and a kernel size of $K = 3$ the required memory is $NK^2M \cdot 32\text{bit} \approx 73.7\text{kB}$. A ternary representation requires only $2 \cdot NK^2M \cdot 1\text{bit} + 2 \cdot 32\text{bit} \approx 4.6\text{kB}$. This corresponds to a $16\times$ compression. As whole channels are deleted by applying entropy-constrained trained ternarization, the effective compression rate is even higher. Using sparse matrix formats like compressed entropy row (CER) or entropy coding techniques can further shrink the storage footprint losslessly.

Ternary networks are not only memory-efficient but also computationally efficient: as wil be shown shown in section 4.3 multiply-accumulate (MAC) operations on ternary weight matrices can be replaced by simple additions and two multiplications only. It is recalled that MACs constitute the fundamental arithmetic of a convolutional layer. In our proposed framework we exploit the benefits of ternary quantization to solve the defined problem of finding the most efficient neural network representation which solves a given task to a specified quality level.

From the assignment cost function C defined in (4.4), a set of equations can be set up. In the ternary case, for each layer l (4.4) yields the following equation system:

$$C_n^{(l)} = d(\mathbf{W}^{(l)}, w_n^{(l)})^2 - \lambda^{(l)} \cdot \log_2 P_n^{(l)} \tag{4.5}$$

$$C_0^{(l)} = d(\mathbf{W}^{(l)}, w_0)^2 - \lambda^{(l)} \cdot \log_2 P_0^{(l)} \tag{4.6}$$

$$C_p^{(l)} = d(\mathbf{W}^{(l)}, w_p^{(l)})^2 - \lambda^{(l)} \cdot \log_2 P_p^{(l)} \tag{4.7}$$

As described above, λ controlls the intensity of the entropy constraint and thus the amount of network sparsification. In order to find an appropriate λ per layer l, we

define

$$\lambda^{(l)} = \gamma \cdot \delta^{(l)} \cdot \lambda^{(l)}_{\max} \tag{4.8}$$

where γ is the gain factor for network sparsity and a global hyperparameter for the whole network. With $\delta^{(l)}$ we formulate a decay of the constraint intensity, taking into account the number of weights per network layer. We want to constrain large layers more than smaller ones because large layers have more redundancy and more capacity. Therefore, for each layer we calculate δ as follows:

$$\delta^{(l)} = \left(\frac{\#\mathbf{W}^{(l)}}{\max\limits_{l=0,1,\dots,L} \#\mathbf{W}^{(l)}} + \varsigma \right) \frac{1}{\varsigma + 1} \tag{4.9}$$
$$\text{with} \quad 0 \leq \varsigma < 1.$$

The ratio in (4.9) describes the number of all parameters of layer l to the number of all parameters of the largest layer to quantize. With a sustain factor ς the decay factor is bound according to $\varsigma \lesssim \delta^{(l)} \leq 1$.
$\lambda^{(l)}_{\max}$ is reached if either

$$\forall_{w \in \mathbf{W}^{(l)}} \quad C_0^{(l)} \leq C_n^{(l)} \tag{4.10}$$

or

$$\forall_{w \in \mathbf{W}^{(l)}} \quad C_0^{(l)} \leq C_p^{(l)}. \tag{4.11}$$

That is when the assignment function would create a binary layer and $\hat{w} \leftarrow \{w_n, w_0, w_p\}$ would be replaced by either $\hat{w} \leftarrow \{w_n, w_0\}$ or $\hat{w} \leftarrow \{w_0, w_p\}$. To avoid this case, we solely apply λ s.t. $\lambda^{(l)} < \lambda^{(l)}_{\max}$. For both inequalities, (4.10) and (4.11), we can insert (4.5)-(4.7) and solve for $\lambda^{(l)}$ which yields

$$\lambda^{(l)}_{\max_n} = \frac{d(w^{(l)}_{\min}, w_0)^2 - d(w^{(l)}_{\min}, w^{(l)}_n)^2}{\log_2 P_0^{(l)} - \log_2 P_n^{(l)}} \tag{4.12}$$

and

$$\lambda^{(l)}_{\max_p} = \frac{d(w^{(l)}_{\max}, w_0)^2 - d(w^{(l)}_{\max}, w^{(l)}_p)^2}{\log_2 P_0^{(l)} - \log_2 P_p^{(l)}} \tag{4.13}$$

with

$$\lambda^{(l)}_{\max} = \min\left(\lambda^{(l)}_{\max_n}, \lambda^{(l)}_{\max_p}\right). \tag{4.14}$$

Having $\lambda_{\max}$ and δ we want to find the largest $\gamma \in (0, 1]$ possible while maintaining model accuracy. For a given γ we execute the assignment A by solving

$$A^{(l)} = \underset{c}{\operatorname{argmin}} \; C_c^{(l)} \quad \text{with} \quad c \in \{n, 0, p\}. \tag{4.15}$$

Figure 4.3 demonstrates the shifting of decision boundaries caused by different intensities of entropy constraints.

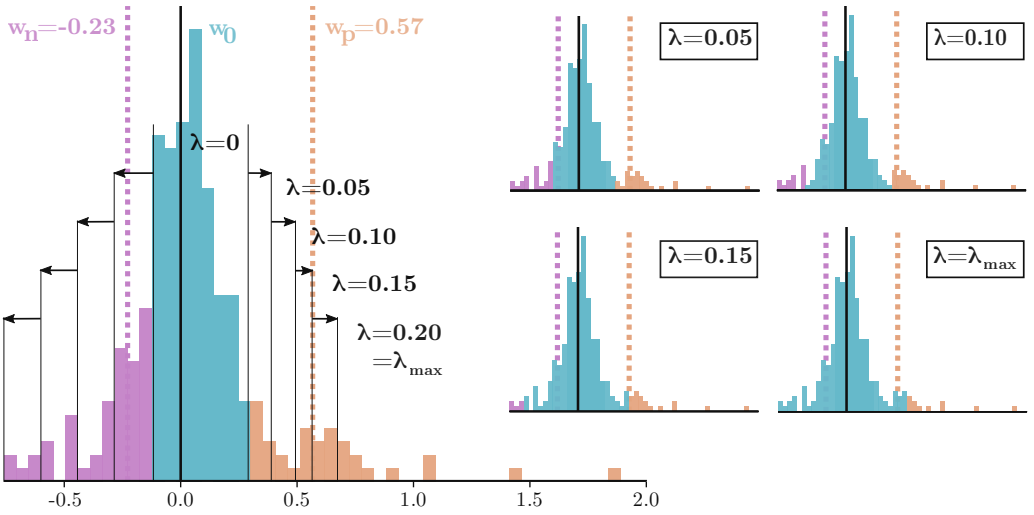

FIGURE 4.3: Principle of entropy-constrained quantization: λ controls the intensity of the constraint. For $\lambda = 0$ the weights are quantized to their nearest neighbor centroids. Here, as initial centroids we use $w_n = 0.3 \cdot \min(\mathbf{W})$ and $w_p = 0.3 \cdot \max(\mathbf{W})$. From the number of assigned weights per cluster, the initial probabilities $P_n = 0.23, P_0 = 0.63, P_p = 0.14$ and information contents $I_n = 2.14, I_0 = 0.67, I_p = 2.79$ result. According to the information content, it is more expensive to delete weights from the w_p-cluster (i.e. assigning them to w_0) than from the w_n-cluster. That is why the shifts of the decision boundaries are comparably small in the positive valued domain when increasing λ. We define λ_{max} as the λ for which one of the two clusters is completely assigned to zero, i.e. a binary quantization results. The histogram data is based on real weight values from the projection convolution of the first MBconv block of our EfficientNet-B1.

As shown in figure 4.3, we initialize the centroids according to

$$w_{n_{\text{ini}}}^{(l)} = |\min(\mathbf{W}^{(l)})| \cdot s \quad \text{and} \quad w_{p_{\text{ini}}}^{(l)} = |\max(\mathbf{W}^{(l)})| \cdot s \tag{4.16}$$

where s determines the initial sparsity. During the subsequent training iterations, the centroids are updated based on the loss of the ternary model. The gradient of a centroid is calculated by taking the sum of all ternary model parameters' gradients which are assigned to the respective centroid:

$$\nabla_{\text{centroids}}^{(l)} = \begin{cases} \frac{\partial \text{loss}}{\partial w_n^{(l)}} = \sum\limits_{w_t \in \mathbf{W}_t^{(l)}} \frac{\partial \text{loss}}{\partial w_t}, & w_t = w_n^{(l)} \\ \frac{\partial \text{loss}}{\partial w_p^{(l)}} = \sum\limits_{w_t \in \mathbf{W}_t^{(l)}} \frac{\partial \text{loss}}{\partial w_t}, & w_t = w_p^{(l)} \end{cases} \tag{4.17}$$

where $\mathbf{W}_t^{(l)}$ characterizes the weight matrix of a ternarized layer. As mentioned in the previous section, the gradients of the quantized network are not only used to update the centroids but also to update a full precision background model which is a copy of the

initial supernet. The gradients of the background model are derived as follows:

$$\nabla^{(l)}_{\text{background}} = \begin{cases} w_n^{(l)} \times \frac{\partial \text{loss}}{\partial w_t}\Big|_{w_t \in \mathbf{W}_t^{(l)}}, & w_t = w_n^{(l)} \\ \frac{\partial \text{loss}}{\partial w_t}\Big|_{w_t \in \mathbf{W}_t^{(l)}}, & w_t = 0 \\ w_p^{(l)} \times \frac{\partial \text{loss}}{\partial w_t}\Big|_{w_t \in \mathbf{W}_t^{(l)}}, & w_t = w_p^{(l)} \end{cases} \tag{4.18}$$

which is copying the gradients of the ternary model for all zero-parameters and scaling the remaining gradients with the associated centroid values. After having updated the background model and the centroids, the parameters of the background model are assigned to the clusters. Figure 4.4 illustrates the parameter and gradient updates of the actual ternary model, the centroids and the full precision background model. At the end of this section we provide a pseudo-code for EC2T, see algorithm 1 and 2.

4.2.2 Special Case: Entropy-Constrained Pruning (ECP)

As depthwise or pointwise convolutions inherently exhibit a small amount of redundancy, it's more difficult to ternarize them without causing a huge drop in accuracy. Especially depthwise convolutions, which avoid inter-channel information processing, seem to be really sensitive to ternarization and pruning. Finding redundant or less important weights in such a trimmed parameter space is challenging.

Therefore, we apply entropy-constrained pruning to the following ternarizing-sensitive EfficientNet layers: depthwise convolutions, Squeeze-and-Excitation convolutions and fully connected output layer. Entropy-constrained pruning (ECP) is identical to the EC2T algorithm but updates only those weight values of the ternary model which would be assigned to the zero-valued cluster. That is, in step 9 of figure 4.4, the weight values are updated only if the assignment cost is the smallest for $w_0 = 0$. The weights which would be assigned to w_n or w_p receive a copy from the weight values of the updated full precision background model (step 6 in figure 4.4).

As can be seen in table 4.3, ECP is applied to layers which do not contribute much to the total number of FLOPs. Thus, the efficiency benefits due to EC2T remain unchanged for the majority of network parameters.

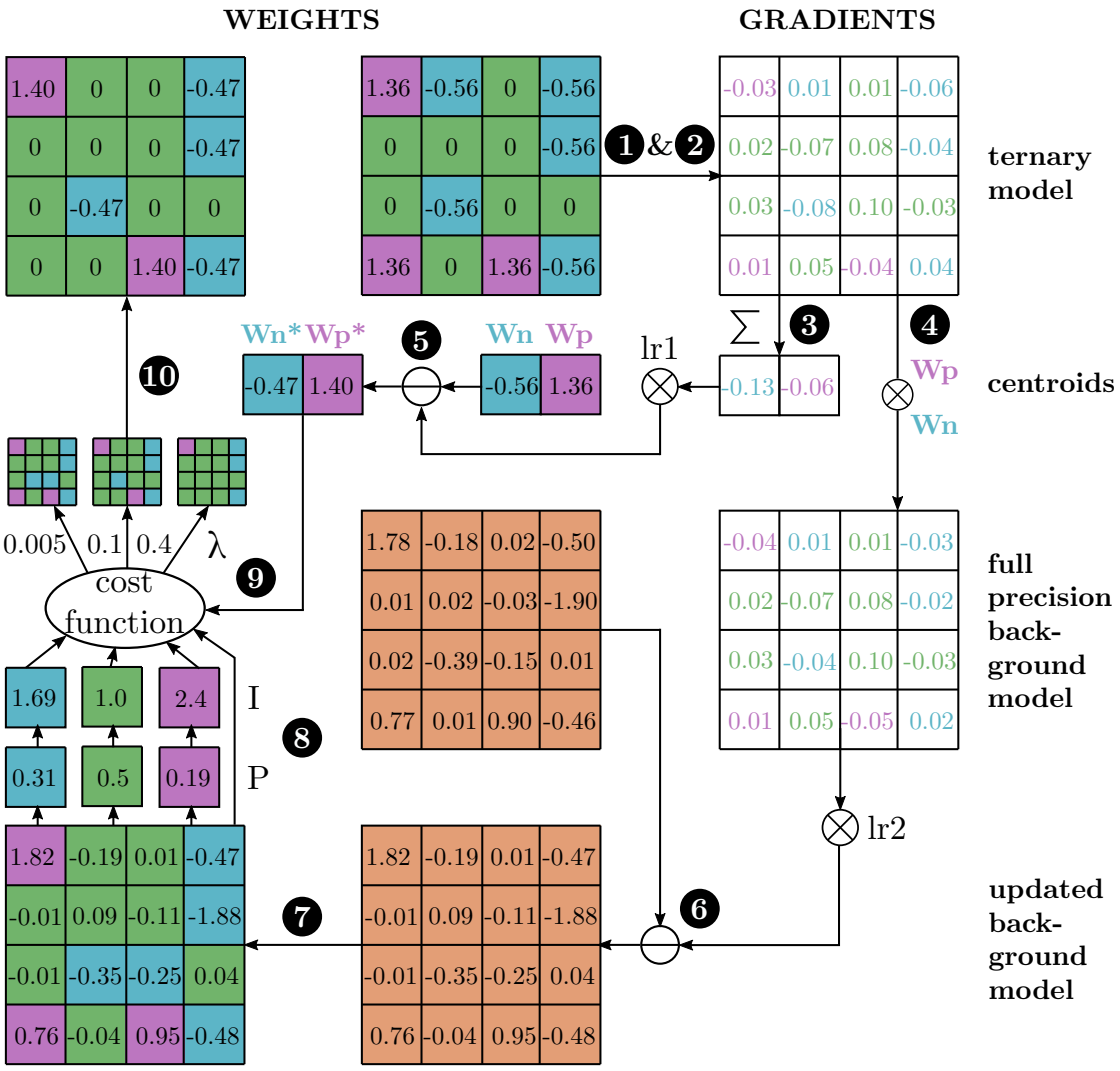

FIGURE 4.4: EC2T: exemplary iteration of a gradient and weight update within a ternary layer. 1) Execute a forward pass using the ternary model; 2) backpropagate the error and calculate gradients of the ternary model; 3) from the ternary model gradients derive the gradients for the centroids which is the sum of the cluster-associated gradients; 4) from the ternary model gradients derive the gradients for the full precision background model which is a scaled version of the ternary model gradients: multiply the w_p-associated gradients with w_p and the w_n-associated gradients with w_n; 5) update the centroids with a learning rate of lr1 (which is 0.7 in this example); 6) update the background model with a learning rate of lr2 (which is 1 in this example); 7) execute a preliminary cluster assignment which is the minimum squared distance of the updated centroids to the updated background model weights; 8) calculate statistics like probabilities P and information contents I of the clusters; 9) calculate the assignment cost for each background model weight and chose the centroid with the smallest cost ; 10) the sparsity of the updated ternary layer depends on the effective λ and, following the example, $\lambda_{\text{eff}} = 0.1$ would delete w_{01} and w_{30} in this iteration. The example also illustrates that the weight w_{22} was almost re-assigned to w_n but λ was chosen too large. In a subsequent iteration the re-assignment of w_{22} might be possible.

TABLE 4.3: EfficientNet-B1 layer types and their contribution according to the total number of parameters and FLOPs

Layer type	**#FLOPs**	**#params**	**EC2T**	**ECP**
conv stem (1×)	26.0M	0.00M		
exp convs (21×)	**520.2M**	**2.23M**	✓	
dw convs (23×)	137.1M	0.27M		✓
SEred convs (23×)	5.2M	0.55M		✓
SEexp convs (23×)	5.3M	0.57M		✓
proj convs (23×)	**575.3M**	**2.45M**	✓	
conv head (1×)	52.8M	0.41M	✓	
FC (1×)	2.6M	**1.28M**		✓
total	1324.5M	7.76M		

Algorithm 1: Entropy-Constrained Trained Ternarization (EC2T) - Initialization

$model(\mathbf{W}) \longleftarrow$ loading pretrained full precision model weights $\mathbf{W}$
$dataloader \longleftarrow$ dataset, data preprocessing
$layers\_to\_quantize(\mathbf{W})$, $unquant\_params(\mathbf{W}_{FP}) \longleftarrow model$ partitioning
$background\_model(\mathbf{W}) \longleftarrow$ copy of $layers\_to\_quantize(\mathbf{W})$
for l in $layers\_to\_quantize$ **do**
  Initialize centroids:
    $w_n^{(l)} \longleftarrow \min(\mathbf{W}^{(l)}) \times s$
    $w_0 \longleftarrow 0$
    $w_p^{(l)} \longleftarrow \max(\mathbf{W}^{(l)}) \times s$
  Initial ternarization of layer weights $\mathbf{W}^{(l)}$:
  for w in $\boldsymbol{W}^{(l)}$ **do**
    $w \longleftarrow \underset{w_c}{\operatorname{argmin}} (w - w_c)^2,\ c \in \{n, 0, p\}$
  end
end
Define Adam optimizers:
  $opt\_ternary \longleftarrow model(layers\_to\_quantize^*,\ unquant\_params)$
  $opt\_centroids \longleftarrow\ [w_n^{(l)}, w_p^{(l)}]_{\forall_l}$
  $opt\_background \longleftarrow background\_model$
Main loop: **Algorithm 2**

Algorithm 2: Entropy-Constrained Trained Ternarization (EC2T) - Main loop

Initialization: **Algorithm 1**
for *epoch* in *num_epochs* **do**
$\hat{y} \longleftarrow model(\mathbf{W}, train\_data)$
$loss \longleftarrow$Cross entropy$(\hat{y}, y)$
$\nabla_{opt\_ternary} = \nabla_{opt\_centroids} = \nabla_{opt\_background} = 0$
$\nabla_{opt\_ternary} = \frac{\partial loss}{\partial \mathbf{W}}$

Derive gradients from ternary model:
for l in *layers_to_quantize* **do**

$$\nabla^{(l)}_{centroids} = \begin{cases} \frac{\partial \text{loss}}{\partial w_n^{(l)}} = \sum_{w_t \in \mathbf{W}_t^{(l)}} \frac{\partial \text{loss}}{\partial w_t}, & w_t = w_n^{(l)} \\ \frac{\partial \text{loss}}{\partial w_p^{(l)}} = \sum_{w_t \in \mathbf{W}_t^{(l)}} \frac{\partial \text{loss}}{\partial w_t}, & w_t = w_p^{(l)} \end{cases}$$

$$\nabla^{(l)}_{background} = \begin{cases} w_n^{(l)} \times \frac{\partial \text{loss}}{\partial w_t}\Big|_{w_t \in \mathbf{W}_t^{(l)}}, & w_t = w_n^{(l)} \\ \frac{\partial \text{loss}}{\partial w_t}\Big|_{w_t \in \mathbf{W}_t^{(l)}}, & w_t = 0 \\ w_p^{(l)} \times \frac{\partial \text{loss}}{\partial w_t}\Big|_{w_t \in \mathbf{W}_t^{(l)}}, & w_t = w_p^{(l)} \end{cases}$$

$\nabla^{(l)}_{layers\_to\_quantize} = 0$
$opt\_centroids^{(l)} \longleftarrow \nabla^{(l)}_{centroids}$
$opt\_background^{(l)} \longleftarrow \nabla^{(l)}_{background}$
$opt\_ternary^{(l)} \longleftarrow \nabla^{(l)}_{layers\_to\_quantize}$
end

Full-precision weight updates:
$[w_n^{*(l)}, w_p^{*(l)}]_{\forall_l} \longleftarrow [w_n^{(l)} - \mu \cdot \partial loss/\partial w_n^{(l)},\ w_p^{(l)} - \mu \cdot \partial loss/\partial w_p^{(l)}]_{\forall_l}$
$background\_model^* \longleftarrow background\_model - \eta \cdot \nabla_{background}$
$unquant\_params^* \longleftarrow unquant\_params - \eta \cdot \nabla_{opt\_ternary}$

Ternary assignment:
for l in $layers\_to\_quantize^*$ **do**
for w in $\boldsymbol{W}^{(l)}$ **do**
$w \longleftarrow \underset{w_c}{\text{argmin}}\,(w - w_c)^2,\ c \in \{n, 0, p\}$
end
$P_c^{(l)} = \#w_c^{(l)} \ / \ \#\mathbf{W}^{(l)},\ c \in \{n, 0, p\}$
$C_c^{(l)} = d(\mathbf{W}^{(l)}, w_c^{(l)})^2 - \lambda^{(l)} \cdot \log_2 P_c^{(l)}$
for w in $\boldsymbol{W}^{(l)}$ **do**
$w \longleftarrow \underset{w_c}{\text{argmin}}\ C_c^{(l)},\ c \in \{n, 0, p\}$
end
end
end

4.3 Efficiency Estimation of Sparse and Ternary CNNs

In order to quantify the memory and compute efficiency of our resulting compressed CNNs, we count the number of all parameters and the number of additions plus multiplications that are required to perform inference on one data sample, i.e. forward passing one image through the network. In the following chapter we clarify the efficiency benefit of utilizing ternary, sparse convolution layers and describe the counting of parameters and FLOPs in detail. For counting parameters and mathematical operations we stick to the scoring rules defined by MicroNet challenge [83] held at NeurIPS conference 2019.

4.3.1 Parameter Storage

In addition to the trainable network parameters, we also count values that are needed to reconstruct the model from sparse matrix formats, i.e. bitmasks or indices. We count a 32 bit parameter as one parameter and quantized parameters of less than 32 bit as a fraction of one parameter. For example, the binary bitmask parameters count as 1/32nd of a parameter.

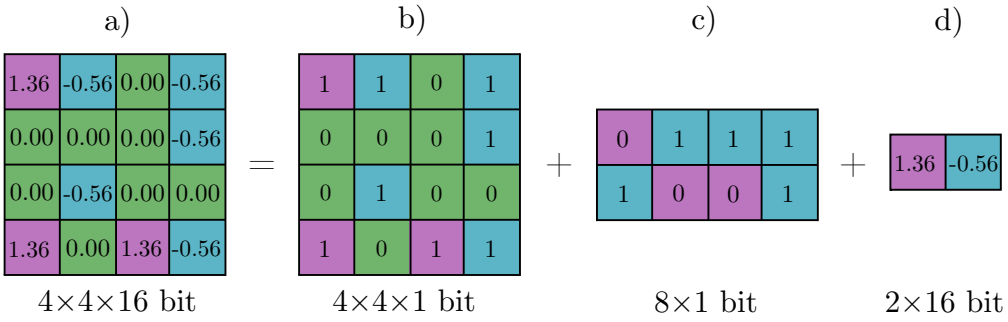

FIGURE 4.5: Efficient storage of sparse, ternary weight matrices.

If no CSR or CER format is applied, a ternary convolution layer of size $\mathcal{N}K^2\mathcal{M}$ consists of two binary bitmasks: one sparse bitmask denoting the location of the centroids and another sign-bitmask linking to the negative or positive valued centroid, respectively (figure 4.5b and 4.5c). They count as $1/32 \times \mathcal{N}K^2\mathcal{M}$ and $1/32 \times \sigma\mathcal{N}K^2\mathcal{M}$ parameters, where $\mathcal{N}$ indicates the number of effective input channels, K the kernel size, $\mathcal{M}$ the number of effective output channels and $\sigma = 1 - \text{sparsity} \leq 1$. With "effective number of channels" we mean the original channel number minus the number of pruned channels due to EC2T. To calculate the layer's sparsity, we exclude the pruned channels from calculation. We add two 16 bit numbers for the actual centroid values which counts as one full parameter (figure 4.5d). In order to account for the BatchNorm layers, we add one 16 bit bias value per effective output channel which counts as $\mathcal{M}/2$ parameters.

4.3.2 Floating Point Operations

In order to calculate the mean number of arithmetic operations per data sample, a 32 bit operation counts as one operation. Multiplication operations on data of less than 32 bit count as a fraction of one operation, where the numerator is the maximum number of bits

in the inputs of the operation and the denominator is 32. For example, a multiplication operation with one 3 bit and one 5 bit input will count as 5/32nd of an operation. For accumulating the products generated by convolutional filter application, we employ a tree adder which is described later in this section. We do not take dynamic activation sparsity into account because we do not take activation storage size into account.

Multiplications and Additions of Sparse Ternary Layers

A great advantage of ternary networks is that the number of multiplications while inference is reduced to only two per filter application, which we will demonstrate in the following. A single kernel application can be thought as a dot product between the kernel weights and the patches of the input image. Dot products of two vectors of size n require n multiplications and $n-1$ additions:

$$\begin{matrix} i_{00} & i_{01} & i_{02} & \cdots \\ i_{10} & i_{11} & i_{12} & \cdots \\ i_{20} & i_{21} & i_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix} * \begin{bmatrix} k_{00} & k_{01} & k_{02} \\ k_{10} & k_{11} & k_{12} \\ k_{20} & k_{21} & k_{22} \end{bmatrix} = i_{00} \cdot k_{00} + i_{01} \cdot k_{01} + \cdots + i_{22} \cdot k_{22} \tag{4.19}$$

with i_{hw} being the image pixels and k_{yx} being the kernel weights. Referring to the above example with $n = 9$, a single 3×3 kernel application requires 17 FLOPs.

In the ternary case, two binary kernel matrices exist, one for pointing at the negative valued centroids w_n and one for pointing at the positive valued centroids w_p. The matrices are applied in two subsequent forward passes:

$$\begin{matrix} i_{00} & i_{01} & i_{02} & \cdots \\ i_{10} & i_{11} & i_{12} & \cdots \\ i_{20} & i_{21} & i_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix} * \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}_{w_n} = (i_{01} + i_{20}) \cdot w_n \tag{4.20}$$

$$\begin{matrix} i_{00} & i_{01} & i_{02} & \cdots \\ i_{10} & i_{11} & i_{12} & \cdots \\ i_{20} & i_{21} & i_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}_{w_p} = (i_{00} + i_{12} + i_{22}) \cdot w_p \tag{4.21}$$

For the above example, which illustrates a ternary kernel application with $\approx 56\%$ sparsity, 2 multiplications and 4 additions are required. Consequently, a sparse ternary convolution layer in general requires

$$2 \times HW\mathcal{M} \quad \text{multiplications} \quad + \quad HW(\sigma\mathcal{N}K^2 - 1)\mathcal{M} \quad \text{additions} \tag{4.22}$$

which results in

$$HW(\sigma\mathcal{N}K^2 + 1)\mathcal{M} \quad \text{total FLOPs} \tag{4.23}$$

where H and W indicate the size of the input feature map, $\sigma = 1 - \text{sparsity} \leq 1$, $\mathcal{N}$ the number of effective input channels, K the kernel size and $\mathcal{M}$ the number of effective output channels.

To clarify the efficiency benefit we calculate the number of FLOPs for a full precision, dense convolution layer, a full precision sparse layer and a ternary sparse layer. The layer properties are $H = 16$, $W = 16$, $N = 64$, $\mathcal{N} = 50$, $M = 64$, $\mathcal{M} = 53$, $K = 3$, $\sigma = 92.45\%$. For the full resolution, dense convolution we calculate the FLOPs as defined in chapter 2.1.2. That is $HW(2NK^2-1)M = 18.86$ MFLOPs. The full resolution, sparse convolution requires $HW(2\sigma\mathcal{N}K^2 - 1)\mathcal{M} = 908$ kFLOPs. Finally, the ternary, sparse convolution requires $HW(\sigma\mathcal{N}K^2 + 1)\mathcal{M} = 474$ kFLOPs.

The example shows that a ternary convolution layer with 92.45% sparsity is $\approx 40\times$ more computationally efficient than the full resolution, dense counterpart. The benefit increases linearly with the number of channels and the amount of sparsity.

As we quantize our input data, activations and weights from full precision (32 bit) to half precision (16 bit), our multiplications count as one half of a operation because we multiply 16 bit activations with 1 bit masks which results in an operation of 16 bit precision. Thus, the mentioned number of multiplications for a ternary layer in equation (4.22) is reduced from $2 \times HW\mathcal{N}\mathcal{M}$ to $HW\mathcal{N}\mathcal{M}$ in our counting. In order to reduce the accumulation cost in (4.22) as well, we incorporate a tree adder which is defined in the following section.

Efficient Accumulation

The intermediate multiplications' output precisions of the convolutional filters depend on the weights' and activations' precisions. As mentioned above, we deal with 16 bit output products. Accumulating these products for one filter application usually requires $NK^2 - 1$ additions of $16 + \lceil \log_2(NK^2) \rceil$ bit precision; note that one filter is of size NK^2 and the input precision is of 16 bit precision. Thus, for usual filter sizes accumulation of 16 bit values requires 23 bit to 30 bit precision. To avoid this, we employ a tree adder that partitions addition operations in ascending levels of input precisions.

As illustrated in figure 4.6, in the first level we execute half of the $NK^2 - 1$ additions at 16 bit precision, separately. Because addition of two 16 bit values yields a 17 bit value, in the subsequent level the partial sums' precision is increased by 1 bit. In the following level, again, half of the remaining additions is executed and so forth until only one addition is left. The precision of the last level's addition amounts to $16 + \lceil \log_2(NK^2) \rceil - 1$ bit. The resulting sum is of $16 + \lceil \log_2(NK^2) \rceil$ bit precision and quantized to half precision before fed forward to the successive layer. Note that, for simplicity, we did not mention sparsity in the tree adder example. In practice we accumulate $\sigma NK^2 - 1$ intermediate output products.

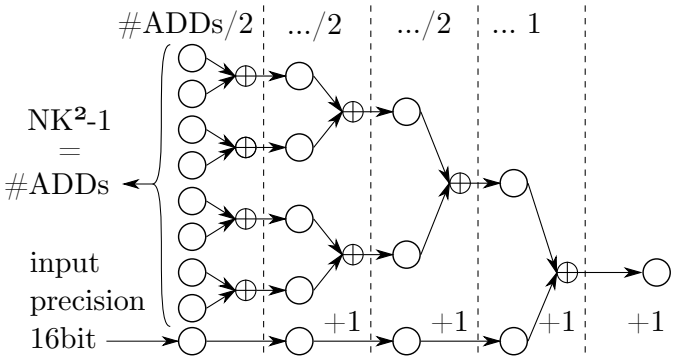

FIGURE 4.6: Tree adder for accumulating output products of a convolution filter of size NK^2, with K being the kernel size and N the number of input channels. The precision of the input products is 16 bit. Going one level deeper increases the operational precision by 1 bit and halves the number of remaining additions.

Power Requirements for FLOPs

For MicroNet challenge 2020 the hosts propose to score operations proportional to their power requirements. For this purpose they suggest to use the frequently-cited energy table by Horowitz, see table 4.4.

TABLE 4.4: Approximate energy cost for various operations in 45nm, 0.9V CMOS processes, from [1].

Operation	**Format**	**Precision**	**Power [pJ]**
Multiplication	Floating-Point	32 bit	3.7
Multiplication	Floating-Point	16 bit	1.1
Multiplication	Integer	32 bit	3.1
Multiplication	Integer	8 bit	0.2
Addition	Floating-Point	32 bit	0.9
Addition	Floating-Point	16 bit	0.4
Addition	Integer	32 bit	0.1
Addition	Integer	8 bit	0.03
Memory read/write	8 kB SRAM	64 bit	10
Memory read/write	DRAM	64 bit	1300-2600

We also use this table to estimate the power consumption of our resulting efficient CNNs. We want to emphasize that our nets require almost no multiplications which are more expensive than additions. Also we aim to reduce power consumption caused by memory reads and writes. Especially, the whole neural network should fit into the internal cache to avoid costly external DRAM access.

Chapter 5

Experiments

In the experiments chapter we first introduce the used datasets. Then we describe aspects of the algorithm's implementation and conduct ablation studies to evaluate our proposed algorithm. Finally, our approach is evaluated on renowned ResNet models and our own models which we found by compound scaling. We present the experimental results and compare them to related works.

5.1 Datasets

In the scope of image classification tasks there exist some commonly used, publicly available datasets which are employed by researchers around the world. This ensures comparability of different neural network models, when applied to the same database. Frequently used datasets are the CIFAR and ImageNet datasets.

The two CIFAR [84] datasets consist of natural images with a resolution of 32×32 pixels. CIFAR-10 contains 10 classes and CIFAR-100 100 classes. The train and test sets contain 50,000 and 10,000 images. Consequently, CIFAR-100 has only 500 training images per class which is challenging.

Whereas CIFAR is a rather small datasets, the ImageNet [85] dataset is a large-scale dataset containing 1.2 million training images and 50,000 test images of 1000 classes. The resolution of the image data is various and in the range of several hundred pixels. In DNN applications the ImageNet data is usually cropped to 224×224 pixels. To achieve higher performances the current trend is to increase the resolution, e.g. to 600×600 pixels, which significantly increases the number of operations of the utilized CNN and thus is less favorable for efficient representations.

5.2 Implementation Aspects

The full code and examples on how to execute compound scaling, EC2T and efficiency scoring can be found at `https://github.com/d-becking/efficientCNNs`. For the implementation we used code from [86–90] which is referenced in the respective headers of the Python files.

5.2.1 Environment

Deep learning frameworks provide basic operations of neural networks computation and efficiently map the computation to high-performance hardware. We used the PyTorch [91] framework for our experiments (version 1.4.0). PyTorch is a recent, flexible framework for deep learning which supports dynamic graph. It uses the NVIDIA CUDA Deep Neural Network library (cuDNN) [92] for GPU acceleration. We use CUDA version 10.1 and cuDNN Version 7603, respectively. As training hardware we used NVIDIA Tesla V100-PCIE-32GB GPUs with driver version 440.33.01 and NVIDIA TITAN V GPUs with driver version 430.40.

5.2.2 Regularization Techniques

A central problem in machine learning is that a model learns to perform well on the training data but poor on new inputs (overfitting). Therefore, strategies designed to reduce the test error and to make the model generalize better, are known as regularization.

Dropout [93] is a technique, where a given number of connections is randomly removed in each training step. In our CIFAR model we use a dropout rate of 0.2 as described in Wide Residual Networks [94]. That is, applying dropout between the two convolutional layers into each residual block. In EfficientNet a dropout rate of 0.2 is applied but only to the final fully connected layer.

Data augmentation is a technique to get around the problem of having limited data samples. Models generalize better when trained on more data. Cropping and horizontal flipping is applied to all datasets. For CIFAR-10 and CIFAR-100 we additionally apply cutout as described in [95]. Cubuk et al. [96] developed an automatic search for well performing data augmentation policies. For CIFAR they found that mostly color-based transformations were picked. Thus, we also use PyTorch's ColorJitter which alters brightness, contrast, saturation and hue depending on a given probability. Prior to data augmentation we normalize the input data.

Weight decay is a L_2 norm penalty which is applied to the weight matrices. While training and compound scaling the CIFAR nets we use a weight decay of 5×10^{-4}, for EfficientNet 1×10^{-4}. During EC2T we apply a weight decay of 5×10^{-6}, but only to the non-quantized parameters, i.e. the input layer, BatchNorm layers, etc.

5.2.3 Training Procedure and Optimization

Step 1: Creating the Supernet. To train the supernets for ImageNet, CIFAR-10 and CIFAR-100 we use stochastic gradient descent (SGD) optimizers with a momentum of 0.9. The initial learning rate is 0.1. For training EfficientNet, the learning rate is decayed by 10 every 30 epochs, training for 90 epochs in total. The other networks are trained for 200-250 epochs using a cosine annealing schedule without restarts [97]. The classification error is quantified with cross entropy loss in all our experiments.

Step 2: Extracting the Optimal Subnet. For the ternarization process, Adam [98] is our optimizer of choice. Since the weights are already settled to a good local

minimum after training the supernet, the learning rate needs to be smaller than training from scratch. But, because quantizing and pruning moves the weights away from the original local minimum, the learning rate should be larger than at the end of step 1 [16]. Thus, we pick a learning rate of 1×10^{-4}. For updating the centroids we chose a learning rate of 1×10^{-5}. We train the ternary assignment plus the centroid values for 20 epochs. Afterwards we fix the assignment and train the centroid values only for further 15 epochs. In the CIFAR networks we apply EC2T to all convolution layers but the first. In order to compress EfficientNet-B1 we apply EC2T to the "exp convs", "proj convs" and "head conv" (cf. figure 4.2 and table 4.3). Afterwards we apply entropy-constrained pruning to the depthwise convolutions, the squeeze-and-excitation convolutions and the fully connected layer with 15 epochs each, while keeping the ternarized and pruned layers fixed. In a final step, we fix all compressed layers and retrain the remaining parameters for 15 epochs.

5.3 Ablation Studies

In this section we validate the effectiveness and performance of our proposed two-step approach. First, we evaluate the accuracy gains due to model compound scaling. In the second subsection, we examine the resulting sparsity pattern when applying EC2T to the supernet and the evolution of the centroids during training.

5.3.1 Model Compound Scaling

In the first stage of model compound scaling we execute a grid search on different depth and width scaling factors s.t. (4.1). The resulting accuracies are listed in table 5.1. We highlighted the best performing scaling factors which are $d = 1.4$ and $w = 1.2$ and $d = 1.2$ and $w = 1.3$. Using this scaling factors we can improve the classification accuracy of CIFAR-100 by 3.46% and of CIFAR-10 by 1.16%, compared to the baseline.

TABLE 5.1: Results of the grid search on depth and width scaling factors for MicroNet-C10 and MicroNet-C100, solving CIFAR-10 and CIFAR-100 respectively.

Model	$d^\star$	$w^\star$	**Acc. C10**†	**Acc. C100**†	**#Params.**
MicroNet-C10/100-0	1.0	1.0	95.02	74.56	0.66M/0.67M
MicroNet-C10/100-1	1.9	1.0	96.02	76.81	1.34M/1.35M
MicroNet-C10/100-2	1.7	1.1	95.98	77.04	1.38M/1.39M
MicroNet-C10/100-3	1.6	1.1	95.64	77.13	1.38M/1.39M
MicroNet-C10/100-4	1.4	1.2	**96.11**	**78.02**	1.52M/1.53M
MicroNet-C10/100-5	1.2	1.3	**96.18**	**77.80**	1.37M/1.38M
MicroNet-C10/100-6	1.0	1.4	95.92	77.17	1.31M/1.32M

† The validation accuracies are reported as mean values of the best five training iterations from 200 total epochs. * depth (d) and width (w) scaling factors.

Subsequently to the grid search we apply compound scaling as defined in (4.3). For our experiments we choose $\phi \in \{1.5, 2, 3, 3.5, 4\}$. It turns out that all compound scaled models outperform the baseline models, see table 5.2. Compound scaling thus further improves the classification accuracy of CIFAR-100 by 3.69% and of CIFAR-10 by 0.96%. Note that the initial accuracy for CIFAR-10 already is at a high level.

TABLE 5.2: Results of compound scaling MicroNet-C10 and MicroNet-C100, solving CIFAR-10 and CIFAR-100 respectively.

Model	ϕ	**Acc. C10**†	**Acc. C100**†	**#Params.**
MicroNet-C10/100-4	1.0	96.11	78.02	1.52M/1.53M
MicroNet-C10/100-4	1.5	96.29	78.47	2.14M/2.15M
MicroNet-C10/100-4	2.0	96.49	79.31	3.26M/3.27M
MicroNet-C10/100-4	3.0	96.76	81.00	6.22M/6.23M
MicroNet-C10/100-4	3.5	**97.02**	**81.47**	8.02M/8.03M
MicroNet-C10/100-4	4.0	97.07	81.49	11.71M/11.72M

† The validation accuracies are reported as mean values of the best five training iterations from 200 total epochs for CIFAR-100 and 250 total epochs for CIFAR-10.

We chose to ternarize the models with $\phi = 3.5$ because we want to achieve at least 80% Top-1 accuracy for CIFAR-100 which will become clear in section 5.4.3. Thus, the models with $\phi = 3.5$ have enough capacity and performance bias for our targeted accuracy. At $\phi = 4$ an upper bound is reached where accuracy gains due to compound scaling diminish and the model starts to overfit on the training data.

5.3.2 Entropy-Constrained Trained Ternarization (EC2T)

In order to validate our main algorithm, entropy-constrained trained ternarization (EC2T), we studied the resulting sparsity patterns of a ternarized network. Figure 5.1 shows the per layer sparsity of our C100-MicroNet. In general, we identify two types of sparsity.

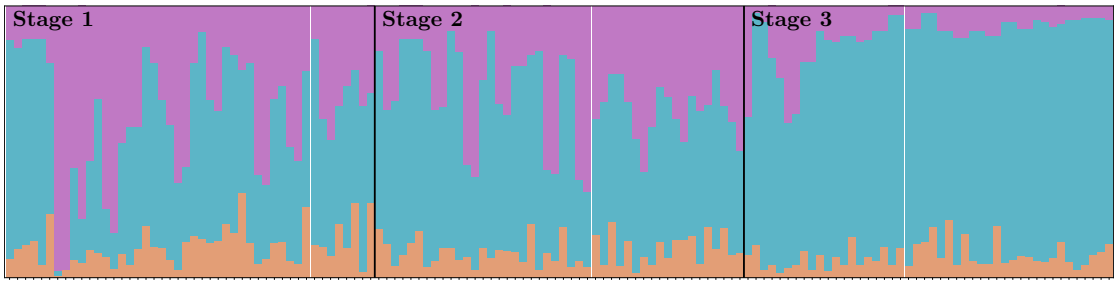

FIGURE 5.1: Sparsity in C100-MicroNet layers due to the application of entropy-constrained trained ternarization. Orange bars depict the relative number of ternary weights, blue bars show the amount of deleted intra-kernel weights and magenta-colored bars depict the number of deleted weights due to channel deletion. Each stack of an orange, blue and magenta bar represents one out of 138 layers, where layer 1 is at the left margin of the plot and layer indices are plotted in ascending order to the right margin.

The first type refers to the deletion of single weights within a kernel (element-wise pruning). The second type refers to the deletion of whole channels (structured pruning).

We do not explicitly delete channels but as we observe a high number of zeros in the resulting weight matrices, we detect "dead" channels in retrospect. For this purpose, in each layer we iterate over all filters M. If all NK^2 filter elements are zero, we delete the according output channel of that layer and the according input channel of the subsequent layer. Thus, channel pruning can be understand as thinning the layers' width and element-wise pruning introduces sparsity to the remaining weights.

As figure 5.1 shows, channel deletion is more frequently in the first stage and second stage. The third and final stage of the network is rather characterized by element-wise pruning, still being somewhat sparser than layers from stage one and two. We conclude, that filters in the deeper layers detect more abstract and unique features from the input data and thus are more important for the network's classification capability. An enlarged version of figure 5.1 including the counts of deleted and remaining weights can be found in the appendix of this work (8.2).

To evaluate the actual progress of the layers' centroids while training, we plotted the assignment distribution and the centroid values of three exemplary layers of the C100-MicroNet during 20 epochs of training (figure 5.2). The chosen three layers are pretty much representative for the ternarization process of most of the layers. We observe

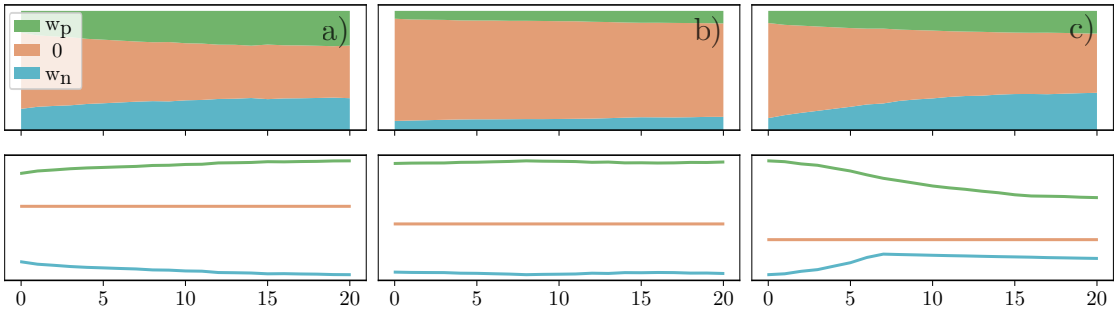

FIGURE 5.2: Evolution of centroid values (bottom row) and distributions (top row) from three different layers of the C10-MicroNet after 20 epochs of EC2T.

that in most of the ternary layers both centroid values either drift apart (figure 5.2a) or aim towards zero (figure 5.2c). Also, we detect layers in which the centroid values rarely change, maintaining the initial values (figure 5.2b). In some layers the values are almost symmetric to zero (figure 5.2b), depending on the underlying full precision weight distribution. Other layers learn rather asymmetric centroids (figure 5.2a, 5.2c).

The assignment distributions, i.e. the number of assigned layer weights to the respective centroids, can also take a symmetric (figure 5.2a, 5.2b) or asymmetric shape (figure 5.2c). Most of the layers maintain the initial distribution due to the entropy constraint (figure 5.2b), others re-assign initially deleted weights. Thus, during training, mistakenly pruned connections are allowed to join a centroid's cluster again. Iterative retraining compensates for the deleted connections, and strengthens connections which are more important in terms of their information content. With EC2T we allow network layers to be architecturally different, and, as Tan et al. [3] and Wu et al. [28] state, layer diversity is crucial for achieving both high accuracy and low latency.

5.4 Evaluation on Datasets

In the following section we report several experiments which were conducted on CIFAR-10, CIFAR-100 and ImageNet datasets. The first subsection aims to compare our method to Trained Ternary Quantization (TTQ) [77] which is the most cited approach in neural network ternarization and kind of a standard in this field. Additionally, we utilized the basic idea of TTQ and extended it in our work; thus a comparison is crucial to evaluate the EC2T algorithm. In the second subsection we deploy EC2T to compress the renowned ResNet-18 and ResNet-20 [20] architectures. These experiments allow comparability of our proposed quantization scheme with ternary quantization methods from related works. In the third subsection we recap the results that we submitted to the MicroNet challenge [83] which was held at NeurIPS conference in December 2019. Also we report further improvements which we implemented beyond the challenge and compare the outcomes with state-of-the-art efficient CNNs.

5.4.1 Comparison With Trained Ternary Quantization (TTQ)

As described in the section's intro, TTQ [77] is a standard in ternary quantization and achieved a precision reduction to ternary values with very little accuracy degradation. We also target a small drop in accuracy but along with an advanced level of sparsity. Compared to our assignment function, which is based on the information content of the centroids and distance measurements, TTQ assigns the full precision weights of the background model by thresholding. First the weights are normalized to a range of $[-1, +1]$ by dividing each weight by the maximum weight value. Then all weights with normalized values larger than t are assigned to the positive valued cluster w_p and all weights with normalized values below $-t$ are assigned to the negative valued cluster w_n. In experiments Zhu et al. [77] set $t = 0.05$ for all layers. Thus, the assignment function itself is symmetric, whereas the learned centroids can be asymmetric. In TTQ, all layers' centroids are initialized with $w_n = -1$ and $w_p = +1$.

In our approach we initialize each layer's centroids based on the layer's weight distribution, i.e. equation (4.17). Then we measure the squared distances from the centroids to the non-normalized full precision background weights. Furthermore, we add our entropy constraint to the distance measurement and assign the background model's weights to the cluster which has the minimal cost according to distance and information content. To compare our method to TTQ we implemented their proposed initialization and assignment, then we linearly increased t to enhance sparsity. The plots in figure 5.3 demonstrate two things: firstly, the observation by Riera et al. [56] that importance heuristics for pruning are irrelevant and only the amount of pruned connections is important may be disproved; secondly, EC2T shows its strength particularly in maintaining accuracy in the super sparse regime beyond 80% of sparsity. In this experiment, the EC2T-generated models achieve ternarization *and* $\sim 95\%$ sparsification of the original model while hurting classification accuracy by only 1%. When comparing the best performing models of this experiment, EC2T achieves its highest accuracy at a sparsity

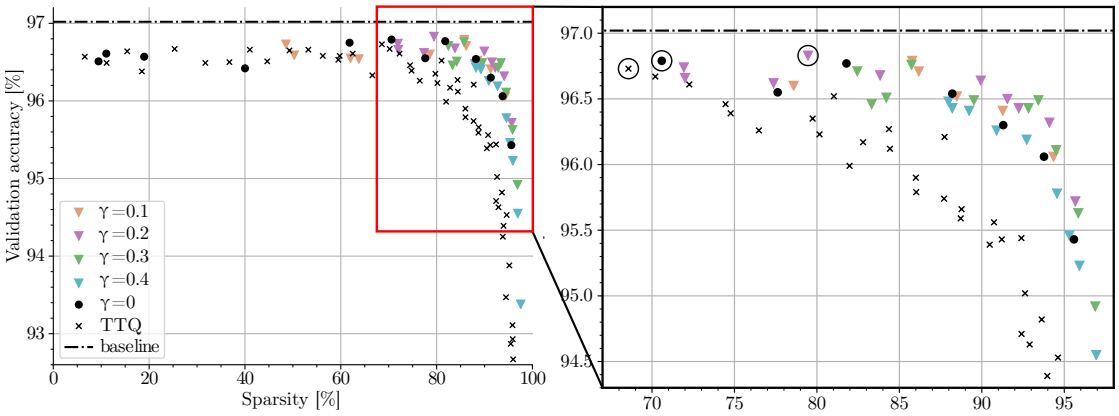

FIGURE 5.3: Performance of the C10-MicroNet, using TTQ vs. our proposal (EC2T). Each data point in this plot represents a ternary neural network model. We linearly increased the threshold factor of TTQ to enhance sparsity. Our approach uses different initial centroid values and gains (γ) for the entropy constraint to find a good trade-off between sparsity and accuracy. Note that $\lambda^{(l)} = \gamma \cdot \delta^{(l)} \cdot \lambda^{(l)}_{\text{max}}$ and $\gamma = 0$ only takes the squared distance into account when applying the assignment function. The circled data points in the right plot highlight the best performing models of each approach, i.e. TTQ, distance-based and distance- plus entropy-based.

level of $\sim 80\%$ whereas TTQ yielded its best model at $\sim 69\%$ sparsity. The observation that more sparse models can outperform less sparse models because of regularization effects is not new and was also reported by Zhu et al. [77].

5.4.2 Comparison With Other Ternary Networks

To make the comparison with other ternary neural networks as fair and comparable as possible, we did not apply any further data augmentation or regularization techniques. We applied our algorithm to the pre-trained ResNet-18 of torchvision's model zoo [1] and ResNet-20 by Yerlan Idelbayev [2] which are public and freely accessible. We ternarized all layers but the input and the output layer.

Table 5.3 summarizes the results. From this table, we can conclude that, compared to existing approaches, EC2T generates state-of-the-art ternary neural networks. Also, we can assume that our model outperforms the other models in terms of efficiency. That is, because the related works do not enhance sparsity explicitly. For instance, Zhu et al. [77] report that their best ternary models exhibit $30 - 50\%$ sparsity. He and Fan [99] initialize their trainable, symmetric decision thresholds at $t^{(l)} = 0.1 \cdot \max(|\mathbf{W}^{(l)}|)$ which introduces a rather small initial sparsity. Also they report issues that those thresholds are pushed close to zero during training.

Furthermore, the table shows that by disabling the entropy constraint, i.e. by setting $\lambda = 0$, the algorithm renders models with rather low sparsity and efficiency (see models with EC2T-1). In contrast, by using $\lambda > 0$, EC2T generates ternary models with increased sparsity that are significantly more efficient in terms of parameter size and

[1] `https://pytorch.org/docs/stable/torchvision/models.html`
[2] `https://github.com/akamaster/pytorch_resnet_cifar10`

mathematical operations. The benefit of efficiency improvement caused by enhanced network sparsity compensates the reasonable accuracy degradation of these models (see EC2T-2, EC2T-3 and EC2T-4 models). Note that 256K of the number of parameters in ResNet-18 belong to the fully connected output layer. An additional application of entropy-constrained pruning (ECP) could further reduce the number of required parameters for model storage. However, for better comparison we do not apply ECP.

TABLE 5.3: EC2T vs state-of-the-art ternary quantization techniques reported for ResNet-20/18 in CIFAR-10/ImageNet datasets.

Model	**Acc. (%)**†	$\frac{\lvert W=0\rvert}{\lvert W\rvert}$ **(%)**‡	**#Params.**	#+	#×	**#FLOPs**
• **ImageNet**						
ResNet-18^{a}	69.75	0.00	11M	1795M	1797M	3592M
EC2T-1 ($\lambda = 0$)^{b}	67.30	26.80	852K	669M	59M	728M
EC2T-2 ($\lambda > 0$)^{c}	67.58	59.00	734K	560M	61M	622M
EC2T-3 ($\lambda > 0$)^{c}	67.26	72.09	686K	528M	57M	585M
EC2T-4 ($\lambda > 0$)^{c}	67.02	75.62	673K	424M	57M	481M
TTQ [77]	66.60	30-50	⊘	⊘	⊘	⊘
ADMM [100]	67.00	⊘	⊘	⊘	⊘	⊘
TGA [99]	66.00	⊘	⊘	⊘	⊘	⊘
• **CIFAR-10**						
ResNet-20^{a}	91.67	0.00	269K	40.6M	40.7M	81.3M
EC2T-1 ($\lambda = 0$)^{b}	91.16	45.17	13.4K	10.6M	0.5M	11.1M
EC2T-2 ($\lambda > 0$)^{c}	91.01	63.90	11.8K	8.0M	0.5M	8.5M
EC2T-3 ($\lambda > 0$)^{c}	90.76	73.26	11.0K	6.1M	0.5M	6.6M
TTQ [77]	91.13	30-50	⊘	⊘	⊘	⊘
TGA [99]	90.39	⊘	⊘	⊘	⊘	⊘
MLQ [101]	90.02	⊘	⊘	⊘	⊘	⊘

^{a} Baseline model. ^{b} EC2T approach without using the entropy constraint (i.e., $\lambda = 0$). ^{c} EC2T approach using the entropy constraint (i.e., $\lambda > 0$). † Top-1 accuracy. ‡ Number of parameters equal to zero over the total number of parameters. ⊘ Not reported by the respective authors.

5.4.3 The MicroNet Challenge

The MicroNet challenge was organized for the first time at the NeurIPS 2019 competition track [3]. Contestants compete to generate the most efficient model that solves a given target task to the specified quality level, i.e. ImageNet to 75% Top-1 accuracy and CIFAR-100 to 80% Top-1 accuracy. The competition is focused on efficient inference, and uses a theoretical metric to measure the efficiency, i.e. by a combination of the number of required math operations and the number of bytes required to store the model parameters. With our EC2T approach we ranked among the top 5 entries for CIFAR-100 and top 10 for ImageNet.

Our submission entries are shown in table 5.4. After quantizing the C100-MicroNet with EC2T, it reaches a sparsity of 90.49% while causing a drop in accuracy of only

[3] `https://nips.cc/Conferences/2019/CompetitionTrack`

1.34% (see C100-MicroNet+EC2T-1). Moreover, the model size and FLOPs are reduced, resulting in 412K and 129M, respectively. These advantages extend to large scale datasets, i.e., ImageNet. For instance, after quantizing the EfficientNet-B1 network with EC2T, a model with 46.33% sparsity is obtained. The accuracy degradation amounts to 3.4% (see EfficientNet-B1+EC2T-1). Note that we upscaled the input image size for EfficientNet-B1 from 240×240 pixels to 256×256 pixels to achieve the target accuracy of 75%.

TABLE 5.4: Results of the MicroNet challenge 2019 in ImageNet and CIFAR-100 datasets. We added our latest improvements and an additional model for CIFAR-10.

Model	**Acc.**†	$\frac{\lvert W=0\rvert}{\lvert W\rvert}$ ‡	**#Params.**	#+	#×	**#FLOPs**
• **ImageNet**						
EfficientNet-B1^{a}	78.43	0.00	7.72M	654M	670M	1324M
+EC2T-1^{b}	75.03	46.33	1.33M	431M	62M	492M
+EC2T-2^{c}	75.05	60.73	1.07M	338M	50M	387M
+Improvements^{d}	-	-	**972K**	212M	50M	**261M**
+ 240×240 input ^{e}	74.35	-	-	192M	44M	237M
MobileNetV2 (d=1.4) [53]	74.70	⊘	6.90M	⊘	⊘	585M*
MobileNetV3 (large) [2]	75.20	⊘	5.40M	⊘	⊘	219M*
• **CIFAR-100**						
C100-MicroNet^{a}	81.47	0.00	8.03M	1243M	1243M	2487M
+EC2T-1^{b}	80.13	90.49	412K	126M	3M	129M
+Improvements^{d}	-	-	**226K**	67M	3M	**71M**
CondenseNet-86 [102]	76.36	⊘	520K	⊘	⊘	65M*
CondenseNet-182 [102]	81.50	⊘	4.20M	⊘	⊘	513M*
• **CIFAR-10**						
C10-MicroNet^{a}	97.02	0.00	8.02M	1243M	1243M	2487M
+EC2T-1^{c}	96.93	80.63	429K	213M	3M	216M
+Improvements^{d}	-	-	257K	114M	3M	117M
+EC2T-2^{c}	96.71	89.99	423K	143M	3M	146M
+Improvements^{d}	-	-	231K	77M	3M	80M
+EC2T-3^{c}	95.87	95.64	295K	72M	3M	75M
+Improvements^{d}	-	-	133K	39M	3M	42M
CondenseNet-86 [102]	95.00	⊘	520K	⊘	⊘	65M*
CondenseNet-182 [102]	96.24	⊘	4.20M	⊘	⊘	513M*

^{a} Baseline model. ^{b} Micronet Challenge submission using the EC2T approach. ^{c} Improved EC2T algorithm. ^{d} After applying tree adder and compressed matrix formats. † Top-1 accuracy in %. ^{e} For our entries with Top-1 accuracy higher than 75% we use 256×256 input image resolution. ‡ Number of parameters equal to zero over the total number of parameters in % . * Reported as Multiply-Additions (MAdds); the number of FLOPs is approximately twice this value. ⊘ Not reported by the respective authors.

Beyond the MicroNet Challenge. We improved the EC2T approach by effectively adjusting the hyperparameter λ in equation (4.4). The adjustment includes a new calculation for λ_{max} and fine-tuning the delay δ by a sustain factor ς as described in equations (4.8)-(4.14). The improved version increased the sparsity of the ternarized EfficientNet-B1 to 60.73% while maintaining similar accuracy (see EfficientNet-B1+EC2T-2).

By employing a tree adder for efficient accumulation and compressed matrix formats for further memory reduction, we achieve additional savings in terms of mathematical operations and storage for both entries (see EfficientNet-B1+EC2T-2+Improvements and C100-MicroNet+EC2T-1+Improvements). The challenge's score is based on the sum of two values: the number of parameters and the mean number of arithmetic operations. Each of this values is normalized by a baseline state-of-the-art model for the respective task, i.e. MobileNetV2 [53] with width 1.4 (6.9M parameters, 1170M math operations) for ImageNet and WideResNet-28-10 [94] (36.5M parameters, 10.49B math operations) for CIFAR-100. In summary, we improved our ImageNet score from 0.6133 to 0.3639 (0.972 M / 6.9 M + 261 M / 1170 M) and our CIFAR-100 score from 0.0236 to 0.0130 (0.226 M / 36.5 M + 71 M / 10.49 B). Note that these values slightly differ from the official MicroNet leaderboard [4] because we improved our counting script as well (which does not alter our rank).

Additionally to our ImageNet and CIFAR-100 entry, we added a ternary and sparse model solving CIFAR-10. This model was already introduced in the previous section to compare with the TTQ approach. We also listed it in table 5.4, in order to establish a new benchmark for this task and to demonstrate the advantages of sparsification (along with ternarization). For the latter we listed three C10-MicroNets with different levels of sparsity and calculated the corresponding efficiency measurements. The number of FLOPs can be reduced by $\sim 21\times$, $\sim 31\times$ and $\sim 59\times$ and the number of parameters by $\sim 31\times$, $\sim 34\times$ and $\sim 60\times$, for sparsity levels of $\sim 80\%$, $\sim 90\%$ and $\sim 95\%$, respectively. The according accuracy degradations amount to $\sim 0.09\%$, $\sim 0.31\%$ and $\sim 1.15\%$.

For the MicroNet challenge 2020 the hosts announce that a new score will take power requirements for mathematical operations into account. For this purpose they suggest to use the frequently-cited energy table by Horowitz (see table 4.4). This further highlights the efficiency of ternary and sparse networks compared to other competing entries, as almost all power-intensive multiplication operations can be discarded. For instance, our improved EfficientNet+EC2T-2 would require 212M $\times$ 0.9pJ $\approx 191\mu$J for additions and 50M $\times$ 3.7pJ $\approx 185\mu$J for multiplications, assuming operations of 32 bit precision. Compared to the EfficientNet-B1 baseline model (654M $\times$ 0.9pJ $\approx 589\mu$J for additions and 670M $\times$ 3.7pJ $\approx 2479\mu$J for multiplications) EC2T improves the power requirements for arithmetic operations by 3068μJ / 376 μJ = $8\times$. Also the power requirements for memory access can be reduced by $\sim 8\times$ because the ternary and spars representation has $\sim 8\times$ less parameters (this estimate ignores memory access for feature map activations).

For a better comparison with respect to state-of-the-art efficient networks, table 5.4 also includes CondenseNet [102], Mobilenet-V2 [53], and Mobilenet-V3 [2]. In the appendix we provide a table with more efficient CNN models solving ImageNet (see table 8.1). Efficient benchmarks for CIFAR-10 and CIFAR-100 are not as commonly reported as for ImageNet which is also evidenced by the normalization model of the CIFAR-100 task in the MicroNet challenge, a WideResNet-28-10 from 2016.

[4] `https://micronet-challenge.github.io/leaderboard.html`

Chapter 6

Conclusion

The aim of this thesis was to design a general framework which generates efficient convolutional neural networks (CNNs) that solve given image classification tasks to a specified quality level. For this purpose, we studied techniques from the field of neural network compression and found that especially neural network quantization and sparsification (pruning) are powerful tools to reduce the computational complexity and the storage footprint of neural networks. In particular, very low-precision neural networks show great compression gains. Moreover, we studied neural architecture search (NAS) methods and concluded that these methods often go hand in hand with an extensive search cost which is, in many cases, not proportionate to the outcome. However, compound model scaling turned out to be a simple NAS method with a reduced search space that can go along with great accuracy gains.

These findings inspired us to our proposed two-step paradigm: 1) utilizing model compound scaling to extend a baseline model until the specified accuracy is reached; 2) applying entropy-constrained trained ternarization (EC2T) to compress the upscaled model by simultaneously quantizing the network parameters to 2 bit (ternarization) and introducing a high level of sparsity due to an entropy constraint.

Following this approach, we rendered efficient models that solve the CIFAR-100 and ImageNet classification tasks to specified classification accuracies of 80% and 75%. We submitted these results to the MicroNet challenge held at NeurIPS conference in December 2019, and ranked among the top 5 and top 10 entries, respectively. In order to score our networks for the challenge, we developed an extensive scoring script which calculates the effective number of parameters and mathematical operations. The script was verified by the challenge's hosts and includes the detection of "dead" channels, as the EC2T algorithm implicitly prunes whole filters. It also takes account of several possibilities to efficiently store ternary and sparse matrices.

Beyond the MicroNet challenge we further investigated our approach in ablation studies and improved the algorithm itself. We compared our method to the standard in ternary quantization, Trained Ternary Quantization (TTQ), and clearly outlined the benefits of EC2T. Utilizing the improved algorithm, compressed matrix formats and a tree adder for efficient accumulation, we significantly enhanced the previous scores. In

another experiment we compressed the well-known ResNet-18 and ResNet-20 networks to compare accuracy degradations due to EC2T with other ternary quantization methods. We concluded that EC2T outperforms related works while producing more efficient networks, as the algorithm explicitly boosts sparsity.

Problems Encountered. One problem encountered was the ternarization of pointwise or, particularly, depthwise convolutions. As these layer types inherently exhibit less redundancy due to an reduced parameter space, further compression quickly results in large accuracy degradation. To address this problem, we designed a novel pruning technique, i.e. entropy-constrained pruning (ECP) which we applied to ternarization-sensitive layers.

Another problem occurred when we compared our results to related works because authors evaluate their results quite differently. In the field of neural network compression usually accuracy degradations vs. compression rates are reported without further measures of efficiency. In works on ternary quantization neither compression rates nor measures of efficiency are reported. Researchers only focus on the objective to ternarize a neural network with a minimal drop in accuracy. Additionally, even in very recent papers, the compressed networks are often obsolete, e.g. LeNet (1998) or AlexNet (2012). Some authors distort their results by making extensive use of regularization techniques or by manipulating the original structure of the networks. In NAS publications the number of mathematical operations is frequently indicated as FLOPs even though the count refers to the number of MACs. Works that embed their resulting networks in efficient hardware accelerators rather report improvements in latency or power consumption which is dependent on the used hardware platform. Consequently, a critical benchmark analysis is required for comparison.

Future Work. As we provide a general framework which is applicable to any PyTorch-build CNN, many other networks, different from EfficientNet, ResNet or our MicroNets, could be ternarized from users to make their applications more efficient. To evaluate the power consumption and latency of our resulting networks, we intend to embed them in specialized hardware accelerators for sparse and ternary matrix computation. Having an efficient hardware solution is not only of interest for research but also in high demand in the industry.

To extend our framework, quantizing feature map activations to less than half precision (16 bit) can probably be realized without degrading the accuracy and thus make further savings in operations. Currently, we only detect pruned channels and discard them for scoring our networks, although they still exist, physically. Therefore a framework extension would be to code a generator that accordingly builds a new model with the reduced number of input and output channels. In further research, ECP as a novel pruning method could be investigated.

Efficient neural networks can open up new and enriching possibilities in a variety of applications, make AI cheaper and more accessible. Through this work, we hope to contribute to this emerging and exciting field of efficient deep learning.

List of Acronyms

AFT	Adaptive Fastfood Transform
AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CER	Compressed Entropy Row
CSR	Compressed Sparse Row
DRAM	Dynamic Random Access Memory
DNN	Deep Neural Networks
EA	Evolutionary Algorithms
ECSQ	Entropy-Constrained Scalar Quantization
FFT	Fast Fourier Transform
FLOP	Floating Point Operation
GPU	Graphics Processing Unit
KD	Knowledge Distillation
MAC	Multiply–Accumulate Operation
ML	Machine Learning
NAS	Neural Architecture Search
PCA	Principal Components Analysis
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SMBO	Sequential Model-Based Optimization
SRAM	Static Random Access Memory
SVD	Singular Value Decomposition
TNN	Ternary Neural Networks
TPU	Tensor Processing Unit
TTQ	Trained Ternary Quantization

Bibliography

[1] Mark Horowitz. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, February 2014. doi: 10.1109/ISSCC.2014.6757323. ISSN: 2376-8606.

[2] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3. *arXiv:1905.02244 [cs]*, November 2019. URL `http://arxiv.org/abs/1905.02244`. arXiv: 1905.02244.

[3] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. *arXiv:1807.11626 [cs]*, July 2018. URL `http://arxiv.org/abs/1807.11626`. arXiv: 1807.11626.

[4] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *arXiv:1808.05377 [cs, stat]*, August 2018. URL `http://arxiv.org/abs/1808.05377`. arXiv: 1808.05377.

[5] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A Survey on Neural Architecture Search. *arXiv:1905.01392 [cs, stat]*, May 2019. URL `http://arxiv.org/abs/1905.01392`. arXiv: 1905.01392.

[6] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, September 2017. doi: 10.23919/FPL.2017.8056850. ISSN: 1946-1488.

[7] Kota Ando, Kodai Ueyoshi, Kentaro Orimo, Haruyoshi Yonekawa, Shimpei Sato, Hiroki Nakahara, Shinya Takamaeda-Yamazaki, Masayuki Ikebe, Tetsuya Asai, Tadahiro Kuroda, and Masato Motomura. BRein Memory: A Single-Chip Binary/Ternary Reconfigurable in-Memory Deep Neural Network Accelerator Achieving 1.4 TOPS at 0.6 W. *IEEE Journal of Solid-State Circuits*, 53(4):983–994, April 2018. ISSN 1558-173X. doi: 10.1109/JSSC.2017.2778702.

[8] Shubham Jain, Sumeet Kumar Gupta, and Anand Raghunathan. TiM-DNN: Ternary in Memory accelerator for Deep Neural Networks. *arXiv:1909.06892 [cs]*, September 2019. URL `http://arxiv.org/abs/1909.06892`. arXiv: 1909.06892.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL `http://www.deeplearningbook.org`.

[10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv:1811.06965 [cs]*, July 2019. URL `http://arxiv.org/abs/1811.06965`. arXiv: 1811.06965.

[11] Qizhe Xie, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. Self-training with Noisy Student improves ImageNet classification. *arXiv:1911.04252 [cs, stat]*, November 2019. URL `http://arxiv.org/abs/1911.04252`. arXiv: 1911.04252.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. *arXiv:1603.05027 [cs]*, July 2016. URL `http://arxiv.org/abs/1603.05027`. arXiv: 1603.05027.

[13] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. SqueezeNext: Hardware-Aware Neural Network Design. *arXiv:1803.10615 [cs]*, August 2018. URL `http://arxiv.org/abs/1803.10615`. arXiv: 1803.10615.

[14] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting Parameters in Deep Learning. *arXiv:1306.0543 [cs, stat]*, October 2014. URL `http://arxiv.org/abs/1306.0543`. arXiv: 1306.0543.

[15] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, December 2017. ISSN 0018-9219, 1558-2256. doi: 10.1109/JPROC.2017.2761740. URL `http://ieeexplore.ieee.org/document/8114708/`.

[16] Song Han. *Efficient Methods and Hardware for Deep Learning*. PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, September 2017. URL `http://purl.stanford.edu/qf934gh3708`.

[17] Hongxu Yin, Bilal Mukadam, Xiaoliang Dai, and Niraj K. Jha. DiabDeep: Pervasive Diabetes Diagnosis based on Wearable Medical Sensors and Efficient Neural Networks. *arXiv:1910.04925 [cs]*, October 2019. URL `http://arxiv.org/abs/1910.04925`. arXiv: 1910.04925.

[18] Alberto Marchisio, Muhammad Abdullah Hanif, Faiq Khalid, George Plastiras, Christos Kyrkou, Theocharis Theocharides, and Muhammad Shafique. Deep

Learning for Edge Computing: Current Trends, Cross-Layer Optimizations, and Open Research Challenges. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 553–559, July 2019. doi: 10.1109/ISVLSI.2019.00105. ISSN: 2159-3477, 2159-3469.

[19] Balázs Csanád Csáji. Approximation with Artificial Neural Networks. *M.Sc. thesis, Faculty of Sciences, Eötvös Loránd University, Hungary*, 2001.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. URL `http://arxiv.org/abs/1512.03385`. arXiv: 1512.03385.

[21] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *arXiv:1707.01083 [cs]*, December 2017. URL `http://arxiv.org/abs/1707.01083`. arXiv: 1707.01083.

[22] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-Excitation Networks. *arXiv:1709.01507 [cs]*, May 2019. URL `http://arxiv.org/abs/1709.01507`. arXiv: 1709.01507.

[23] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *arXiv:1611.01578 [cs]*, February 2017. URL `http://arxiv.org/abs/1611.01578`. arXiv: 1611.01578.

[24] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. *arXiv:1611.02167 [cs]*, March 2017. URL `http://arxiv.org/abs/1611.02167`. arXiv: 1611.02167.

[25] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. *arXiv:1707.07012 [cs, stat]*, April 2018. URL `http://arxiv.org/abs/1707.07012`. arXiv: 1707.07012.

[26] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient Neural Architecture Search via Parameter Sharing. *arXiv:1802.03268 [cs, stat]*, February 2018. URL `http://arxiv.org/abs/1802.03268`. arXiv: 1802.03268.

[27] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *arXiv:1905.11946 [cs, stat]*, May 2019. URL `http://arxiv.org/abs/1905.11946`. arXiv: 1905.11946.

[28] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. *arXiv:1812.03443 [cs]*, May 2019. URL `http://arxiv.org/abs/1812.03443`. arXiv: 1812.03443.

[29] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *arXiv:1812.00332 [cs, stat]*, February 2019. URL `http://arxiv.org/abs/1812.00332`. arXiv: 1812.00332.

[30] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. *arXiv:1806.09055 [cs, stat]*, April 2019. URL `http://arxiv.org/abs/1806.09055`. arXiv: 1806.09055.

[31] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: Stochastic Neural Architecture Search. *arXiv:1812.09926 [cs, stat]*, January 2019. URL `http://arxiv.org/abs/1812.09926`. arXiv: 1812.09926.

[32] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-Path NAS: Designing Hardware-Efficient ConvNets in less than 4 Hours. *arXiv:1904.02877 [cs, stat]*, April 2019. URL `http://arxiv.org/abs/1904.02877`. arXiv: 1904.02877.

[33] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized Evolution for Image Classifier Architecture Search. *arXiv:1802.01548 [cs]*, February 2019. URL `http://arxiv.org/abs/1802.01548`. arXiv: 1802.01548.

[34] Xiangxiang Chu, Bo Zhang, and Ruijun Xu. MoGA: Searching Beyond MobileNetV3. *arXiv:1908.01314 [cs, stat]*, October 2019. URL `http://arxiv.org/abs/1908.01314`. arXiv: 1908.01314.

[35] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring Randomly Wired Neural Networks for Image Recognition. *arXiv:1904.01569 [cs]*, April 2019. URL `http://arxiv.org/abs/1904.01569`. arXiv: 1904.01569.

[36] Han Cai, Chuang Gan, and Song Han. Once for All: Train One Network and Specialize it for Efficient Deployment. *arXiv:1908.09791 [cs, stat]*, August 2019. URL `http://arxiv.org/abs/1908.09791`. arXiv: 1908.09791.

[37] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. *arXiv:1712.00559 [cs, stat]*, July 2018. URL `http://arxiv.org/abs/1712.00559`. arXiv: 1712.00559.

[38] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric Xing. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. *arXiv:1802.07191 [cs, stat]*, March 2019. URL `http://arxiv.org/abs/1802.07191`. arXiv: 1802.07191.

[39] Renato Negrinho and Geoff Gordon. DeepArchitect: Automatically Designing and Training Deep Architectures. *arXiv:1704.08792 [cs, stat]*, April 2017. URL `http://arxiv.org/abs/1704.08792`. arXiv: 1704.08792.

[40] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A Survey of Model Compression and Acceleration for Deep Neural Networks. *arXiv:1710.09282 [cs]*, September 2019. URL http://arxiv.org/abs/1710.09282. arXiv: 1710.09282.

[41] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for Deep Neural Networks. *arXiv:1507.06149 [cs]*, July 2015. URL http://arxiv.org/abs/1507.06149. arXiv: 1507.06149.

[42] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990. URL http://papers.nips.cc/paper/250-optimal-brain-damage.pdf.

[43] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. *arXiv:1611.06440 [cs, stat]*, November 2016. URL http://arxiv.org/abs/1611.06440. arXiv: 1611.06440.

[44] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning Efficient Convolutional Networks through Network Slimming. *arXiv:1708.06519 [cs]*, August 2017. URL http://arxiv.org/abs/1708.06519. arXiv: 1708.06519.

[45] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *ACM Computing Surveys*, 52(2):1–39, May 2019. ISSN 03600300. doi: 10.1145/3309551. URL http://dl.acm.org/citation.cfm?doid=3320149.3309551.

[46] Thomas Wiegand and Heiko Schwarz. Source Coding: Part I of Fundamentals of Source and Video Coding. *Foundations and Trends® in Signal Processing*, 4 (1-2):1–222, 2010. ISSN 1932-8346, 1932-8354. doi: 10.1561/2000000010. URL http://www.nowpublishers.com/article/Details/SIG-010.

[47] Claude E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. URL http://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf.

[48] Simon Wiedemann, Heiner Kirchoffer, Stefan Matlage, Paul Haase, Arturo Marban, Talmaj Marinc, David Neumann, Tung Nguyen, Ahmed Osman, Detlev Marpe, Heiko Schwarz, Thomas Wiegand, and Wojciech Samek. DeepCABAC: A Universal Compression Algorithm for Deep Neural Networks. *arXiv:1907.11900 [cs, math]*, July 2019. URL http://arxiv.org/abs/1907.11900. arXiv: 1907.11900.

[49] Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Compact and Computationally Efficient Representation of Deep Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–14, 2019. ISSN 2162-237X. doi: 10.1109/TNNLS.2019.2910073.

[50] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs]*, November 2016. URL `http://arxiv.org/abs/1602.07360`. arXiv: 1602.07360.

[51] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. *arXiv:1807.11164 [cs]*, July 2018. URL `http://arxiv.org/abs/1807.11164`. arXiv: 1807.11164.

[52] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861 [cs]*, April 2017. URL `http://arxiv.org/abs/1704.04861`. arXiv: 1704.04861.

[53] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv:1801.04381 [cs]*, January 2018. URL `http://arxiv.org/abs/1801.04381`. arXiv: 1801.04381.

[54] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. *arXiv:1804.03230 [cs]*, September 2018. URL `http://arxiv.org/abs/1804.03230`. arXiv: 1804.03230.

[55] Boyu Zhang, Azadeh Davoodi, and Yu Hen Hu. Efficient Inference of CNNs via Channel Pruning. *arXiv:1908.03266 [cs]*, August 2019. URL `http://arxiv.org/abs/1908.03266`. arXiv: 1908.03266.

[56] Marc Riera, Jose-Maria Arnau, and Antonio Gonzalez. (Pen-) Ultimate DNN Pruning. *arXiv:1906.02535 [cs, stat]*, June 2019. URL `http://arxiv.org/abs/1906.02535`. arXiv: 1906.02535.

[57] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured Pruning of Deep Convolutional Neural Networks. *arXiv:1512.08571 [cs, stat]*, December 2015. URL `http://arxiv.org/abs/1512.08571`. arXiv: 1512.08571.

[58] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Neural Networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2074–2082. Curran Associates, Inc., 2016. URL `http://papers.nips.cc/paper/6504-learning-structured-sparsity-in-deep-neural-networks.pdf`.

[59] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient ConvNets. *arXiv:1608.08710 [cs]*, March 2017. URL `http://arxiv.org/abs/1608.08710`. arXiv: 1608.08710.

[60] Yihui He, Xiangyu Zhang, and Jian Sun. Channel Pruning for Accelerating Very Deep Neural Networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, Venice, October 2017. IEEE. ISBN 978-1-5386-1032-9. doi: 10.1109/ICCV.2017.155. URL `http://ieeexplore.ieee.org/document/8237417/`.

[61] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. *arXiv:1608.04493 [cs]*, November 2016. URL `http://arxiv.org/abs/1608.04493`. arXiv: 1608.04493.

[62] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. *arXiv:1412.6553 [cs]*, April 2015. URL `http://arxiv.org/abs/1412.6553`. arXiv: 1412.6553.

[63] Dongsoo Lee, Se Jung Kwon, Byeongwook Kim, and Gu-Yeon Wei. Learning Low-Rank Approximation for CNNs. *arXiv:1905.10145 [cs, stat]*, May 2019. URL `http://arxiv.org/abs/1905.10145`. arXiv: 1905.10145.

[64] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization. *arXiv:1511.06067 [cs, stat]*, February 2016. URL `http://arxiv.org/abs/1511.06067`. arXiv: 1511.06067.

[65] Jimmy Ba and Rich Caruana. Do Deep Nets Really Need to be Deep? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2654–2662. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5484-do-deep-nets-really-need-to-be-deep.pdf`.

[66] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [cs, stat]*, March 2015. URL `http://arxiv.org/abs/1503.02531`. arXiv: 1503.02531.

[67] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. Structured Transforms for Small-Footprint Deep Learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3088–3096. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5869-structured-transforms-for-small-footprint-deep-learning.pdf`.

[68] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang,

Qinru Qiu, Xue Lin, and Bo Yuan. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-CirculantWeight Matrices. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*, pages 395–408, 2017. doi: 10.1145/3123939.3124552. URL `http://arxiv.org/abs/1708.08917`. arXiv: 1708.08917.

[69] Sander Dieleman, Jeffrey De Fauw, and Koray Kavukcuoglu. Exploiting Cyclic Symmetry in Convolutional Neural Networks. *arXiv:1602.02660 [cs]*, May 2016. URL `http://arxiv.org/abs/1602.02660`. arXiv: 1602.02660.

[70] Shuangfei Zhai, Yu Cheng, Weining Lu, and Zhongfei Zhang. Doubly Convolutional Neural Networks. *arXiv:1610.09716 [cs]*, October 2016. URL `http://arxiv.org/abs/1610.09716`. arXiv: 1610.09716.

[71] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. *arXiv:1502.02551 [cs, stat]*, February 2015. URL `http://arxiv.org/abs/1502.02551`. arXiv: 1502.02551.

[72] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv:1606.06160 [cs]*, February 2018. URL `http://arxiv.org/abs/1606.06160`. arXiv: 1606.06160.

[73] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *arXiv:1511.00363 [cs]*, April 2016. URL `http://arxiv.org/abs/1511.00363`. arXiv: 1511.00363.

[74] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]*, March 2016. URL `http://arxiv.org/abs/1602.02830`. arXiv: 1602.02830.

[75] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv:1603.05279 [cs]*, August 2016. URL `http://arxiv.org/abs/1603.05279`. arXiv: 1603.05279.

[76] Fengfu Li, Bo Zhang, and Bin Liu. Ternary Weight Networks. *arXiv:1605.04711 [cs]*, November 2016. URL `http://arxiv.org/abs/1605.04711`. arXiv: 1605.04711.

[77] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. *arXiv:1612.01064 [cs]*, December 2016. URL `http://arxiv.org/abs/1612.01064`. arXiv: 1612.01064.

[78] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]*, October 2015. URL `http://arxiv.org/abs/1510.00149`. arXiv: 1510.00149.

[79] Frederick Tung and Greg Mori. CLIP-Q: Deep Network Compression Learning by In-parallel Pruning-Quantization. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, Salt Lake City, UT, June 2018. IEEE. ISBN 978-1-5386-6420-9. doi: 10.1109/CVPR.2018.00821. URL `https://ieeexplore.ieee.org/document/8578919/`.

[80] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the Limit of Network Quantization. *arXiv:1612.01543 [cs]*, December 2016. URL `http://arxiv.org/abs/1612.01543`. arXiv: 1612.01543.

[81] Simon Wiedemann, Arturo Marban, Klaus-Robert Müller, and Wojciech Samek. Entropy-Constrained Training of Deep Neural Networks. *arXiv:1812.07520 [cs, stat]*, December 2018. URL `http://arxiv.org/abs/1812.07520`. arXiv: 1812.07520.

[82] Dongyoon Han, Jiwhan Kim, and Junmo Kim. Deep Pyramidal Residual Networks. *arXiv:1610.02915 [cs]*, October 2016. URL `http://arxiv.org/abs/1610.02915`. arXiv: 1610.02915.

[83] Trevor Gale, Erich Elsen, Sara Hooker, Olivier Temam, Scott Gray, Jongsoo Park, Cliff Young, Evci Utku, Niki Parmar, and Ashish Vaswani. MicroNet Challenge hosted at NeurIPS 2019, February 2020. URL `https://micronet-challenge.github.io/`.

[84] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. page 60, April 2009.

[85] Jia Deng, Wei Dong, Richard Socher, Li-Ji Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[86] Luke Melas-Kyriazi. lukemelas/EfficientNet-PyTorch, December 2019. URL `https://github.com/lukemelas/EfficientNet-PyTorch`. original-date: 2019-05-30T05:24:11Z.

[87] Dan Antoshchenko. TropComplique/trained-ternary-quantization, December 2019. URL `https://github.com/TropComplique/trained-ternary-quantization`. original-date: 2017-08-21T19:20:51Z.

[88] Yerlan Idelbayev. akamaster/pytorch_resnet_cifar10, December 2019. URL `https://github.com/akamaster/pytorch_resnet_cifar10`. original-date: 2018-01-15T09:50:56Z.

[89] Vithursan Thangarasa. uoguelph-mlrg/Cutout, December 2019. URL `https://github.com/uoguelph-mlrg/Cutout`. original-date: 2017-11-16T18:59:54Z.

[90] Utku Evci. google-research/micronet_challenge at master · google-research/google-research, December 2019. URL `https://github.com/google-research/google-research/tree/master/micronet_challenge`.

[91] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[92] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]*, December 2014. URL `http://arxiv.org/abs/1410.0759`. arXiv: 1410.0759.

[93] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

[94] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. *arXiv:1605.07146 [cs]*, May 2016. URL `http://arxiv.org/abs/1605.07146`. arXiv: 1605.07146.

[95] Terrance DeVries and Graham W. Taylor. Improved Regularization of Convolutional Neural Networks with Cutout. *arXiv:1708.04552 [cs]*, August 2017. URL `http://arxiv.org/abs/1708.04552`. arXiv: 1708.04552.

[96] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. AutoAugment: Learning Augmentation Policies from Data. *arXiv:1805.09501 [cs, stat]*, May 2018. URL `http://arxiv.org/abs/1805.09501`. arXiv: 1805.09501.

[97] Ilya Loshchilov and Frank Hutter. SGDR: Stochastic Gradient Descent with Warm Restarts. *arXiv:1608.03983 [cs, math]*, May 2017. URL `http://arxiv.org/abs/1608.03983`. arXiv: 1608.03983.

[98] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017. URL `http://arxiv.org/abs/1412.6980`. arXiv: 1412.6980.

[99] Zhezhi He and Deliang Fan. Simultaneously Optimizing Weight and Quantizer of Ternary Neural Network using Truncated Gaussian Approximation. *arXiv:1810.01018 [cs, stat]*, October 2018. URL `http://arxiv.org/abs/1810.01018`. arXiv: 1810.01018.

[100] Cong Leng, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely Low Bit Neural Network: Squeeze the Last Bit Out with ADMM. *arXiv:1707.09870 [cs]*, September 2017. URL `http://arxiv.org/abs/1707.09870`. arXiv: 1707.09870.

[101] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. Deep Neural Network Compression with Single and Multiple Level Quantization. *arXiv:1803.03289 [cs, stat]*, December 2018. URL `http://arxiv.org/abs/1803.03289`. arXiv: 1803.03289.

[102] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. CondenseNet: An Efficient DenseNet using Learned Group Convolutions. *arXiv:1711.09224 [cs]*, June 2018. URL `http://arxiv.org/abs/1711.09224`. arXiv: 1711.09224.

[103] Babak Hassibi, David G. Stork, and Gregory Wolff. Optimal Brain Surgeon: Extensions and performance comparisons. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 263–270. Morgan-Kaufmann, 1994. URL `http://papers.nips.cc/paper/749-optimal-brain-surgeon-extensions-and-performance-comparisons.pdf`.

[104] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. *arXiv:1506.02626 [cs]*, June 2015. URL `http://arxiv.org/abs/1506.02626`. arXiv: 1506.02626.

[105] Hao Zhou, Jose M. Alvarez, and Fatih Porikli. Less Is More: Towards Compact CNNs. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, volume 9908, pages 662–677. Springer International Publishing, Cham, 2016. ISBN 978-3-319-46492-3 978-3-319-46493-0. doi: 10.1007/978-3-319-46493-0_40. URL `http://link.springer.com/10.1007/978-3-319-46493-0_40`.

[106] Yu Cheng, Felix X. Yu, Rogerio S. Feris, Sanjiv Kumar, Alok Choudhary, and Shih-Fu Chang. An exploration of parameter redundancy in deep networks with circulant projections. *arXiv:1502.03436 [cs]*, October 2015. URL `http://arxiv.org/abs/1502.03436`. arXiv: 1502.03436.

[107] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. *arXiv:1404.0736 [cs]*, June 2014. URL `http://arxiv.org/abs/1404.0736`. arXiv: 1404.0736.

[108] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. *arXiv:1511.06530 [cs]*, February 2016. URL `http://arxiv.org/abs/1511.06530`. arXiv: 1511.06530.

[109] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Deep3: Leveraging three levels of parallelism for efficient Deep Learning. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. doi: 10.1145/3061639.3062225. ISSN: null.

[110] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and Improving Convolutional Neural Networks via Concatenated Rectified Linear Units. *arXiv:1603.05201 [cs]*, July 2016. URL `http://arxiv.org/abs/1603.05201`. arXiv: 1603.05201.

[111] Edward H. Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. LogNet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904, March 2017. doi: 10.1109/ICASSP.2017.7953288. ISSN: 2379-190X.

[112] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv:1604.03168 [cs]*, October 2016. URL `http://arxiv.org/abs/1604.03168`. arXiv: 1604.03168.

[113] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing Neural Networks with the Hashing Trick. *arXiv:1504.04788 [cs]*, April 2015. URL `http://arxiv.org/abs/1504.04788`. arXiv: 1504.04788.

[114] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing Deep Convolutional Networks using Vector Quantization. *arXiv:1412.6115 [cs]*, December 2014. URL `http://arxiv.org/abs/1412.6115`. arXiv: 1412.6115.

Appendix

8.1 State-of-the-Art efficient CNNs for ImageNet

Table 8.1: ImageNet performance of efficient CNNs from related work.

Model	Top-1 acc (%)	#params	#OPs†	approach
2.0-SqueezeNext-23v5 [13]	67.4	3.2M	708M	manual
ShuffleNetV2 1x[51]	69.4	2.3M	146M	manual
CondenseNet (G=C=8) [102]	71.0	2.9M	274M	manual
MobileNetV2 [53]	72.0	3.4M	300M	manual
SNAS [31]	72.7	4.3M	522M	DNAS
FBNet-A [28]	73.0	4.3M	249M	DNAS
MobileNetV3-Large 0.75[2]	73.3	4.0M	155M	RL & NA
CondenseNet (G=C=4) [102]	73.8	4.8M	529M	manual
NASNet-A [25]	74.0	5.3M	564M	RL
FBNet-B [28]	74.1	4.5M	295M	DNAS
PNASNet-5 (N=3, F=54) [37]	74.2	5.1M	588M	SMBO
AmoebaNet-A (4, 50) [33]	74.5	5.1M	555M	EA
MobileNetV2 (d=1.4) [53]	74.7	6.9M	585M	manual
RandWire-WS [35]	74.7	5.6M	583M	random
FBNet-C [28]	74.9	5.5M	375M	DNAS
ShuffleNetV2 2x [51]	74.9	7.4M	591M	manual
AmoebaNet-C (4, 44) [33]	75.1	5.1M	535M	EA
ProxylessNAS (GPU) [29]	75.1	7.1M	465M	DNAS
MobileNetV3-Large [2]	75.2	5.4M	219M	RL & NA
MnasNet-A1 [3]	75.2	3.9M	312M	RL
MoGA-C [34]	75.3	5.4M	221M	EA
MnasNet-A2 [3]	75.6	4.8M	340M	RL
MoGA-A [34]	75.9	5.1M	304M	EA
EfficientNet-B0 [27]	76.3	5.3M	390M	RL & MS
MnasNet-A3 [3]	76.7	5.2M	403M	RL
EfficientNet-B1 [27]	78.8	7.8M	700M	RL & MS

† Note that the number of operations is listed as reported in the respective works. This number mostly refers to the number of MACs, also called Multiply-Adds (MAdds) and often is mistakenly reported as #FLOPs which is, as shown in this thesis, approximately twice the number of MACs.

8.2 Sparsity in the C100-MicroNet

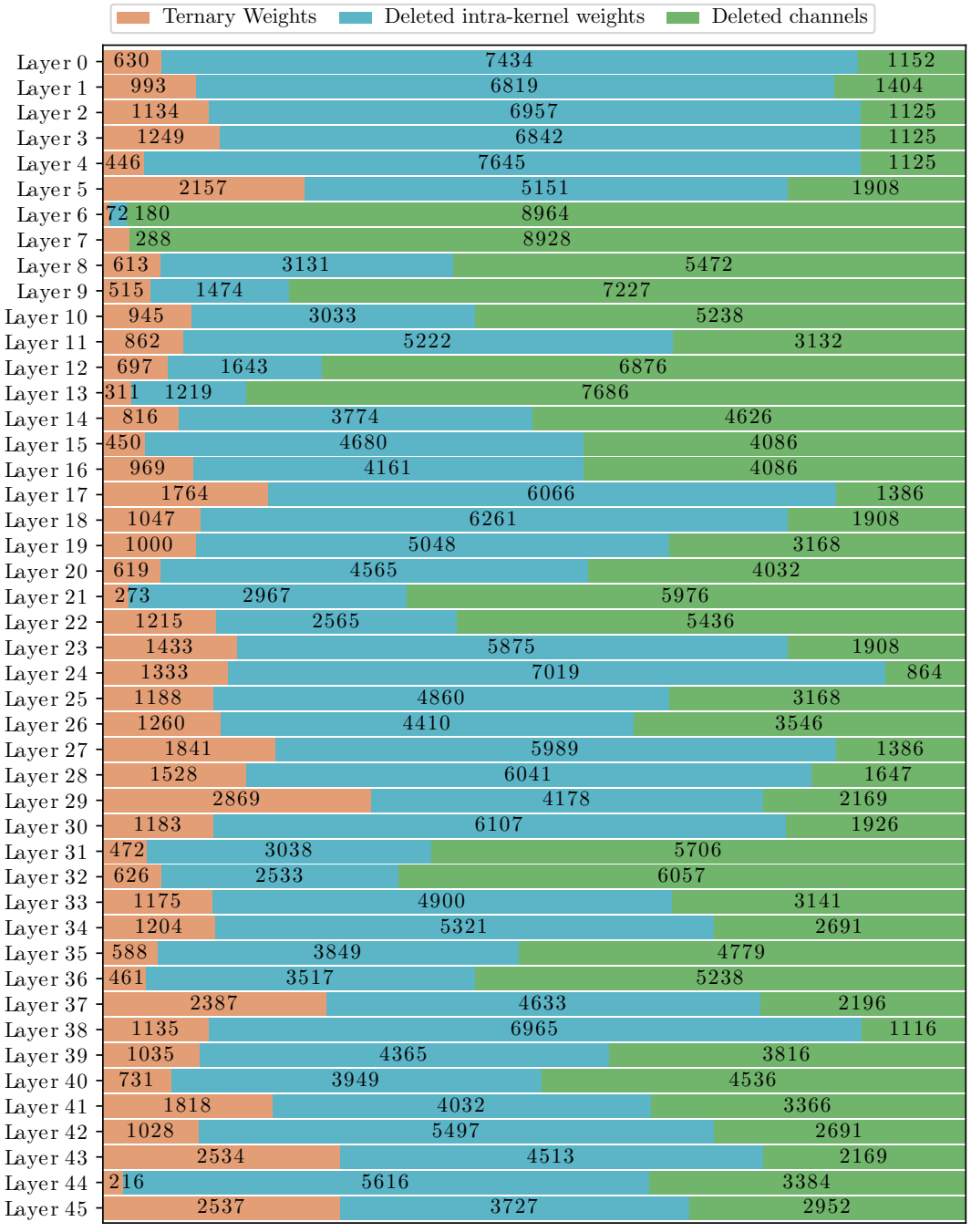

FIGURE 8.1: Per layer sparsity and number of deleted weights due to implicit channel pruning of C100-MicroNet stage one after applying EC2T.

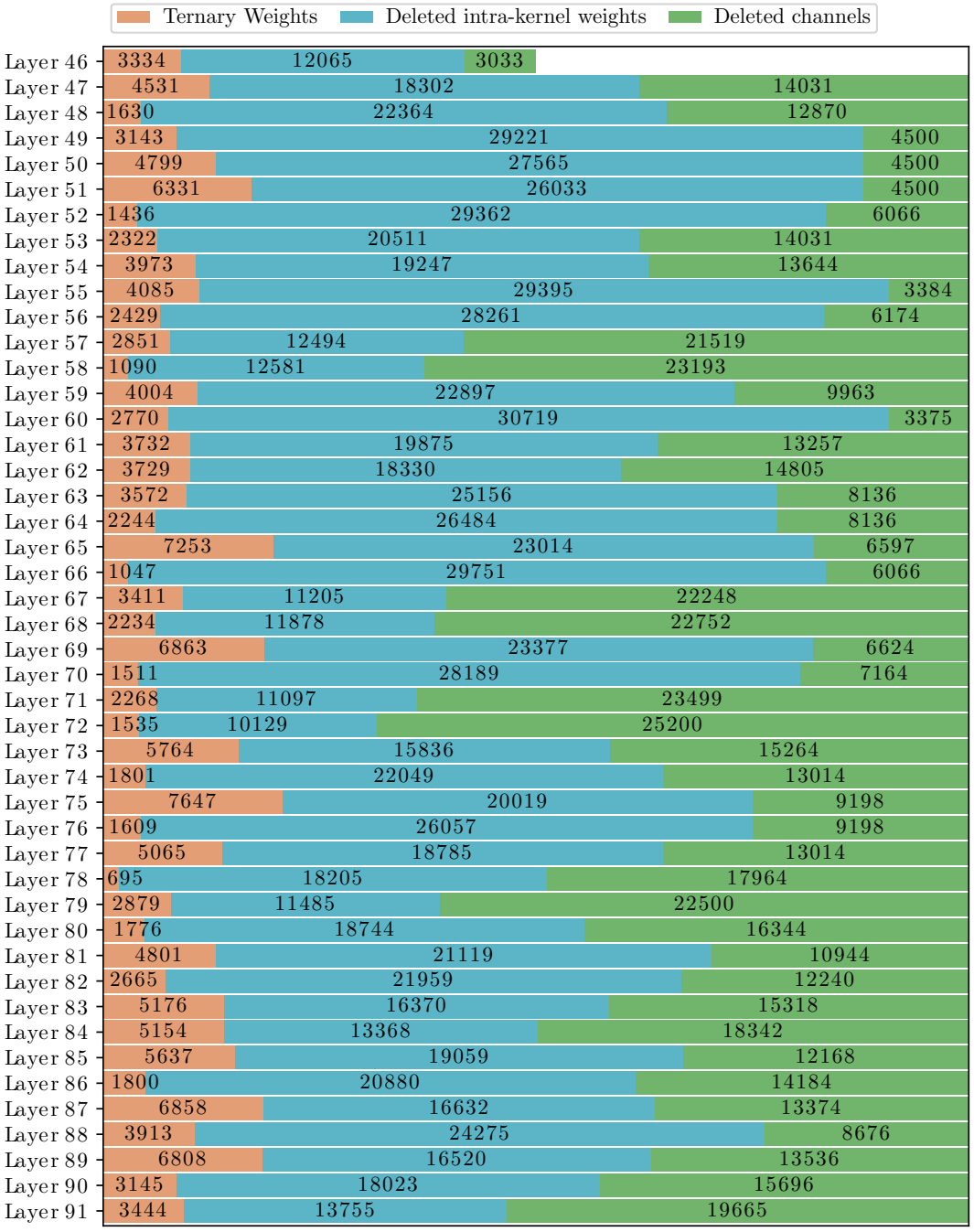

FIGURE 8.2: Per layer sparsity and number of deleted weights due to implicit channel pruning of C100-MicroNet stage two after applying EC2T.

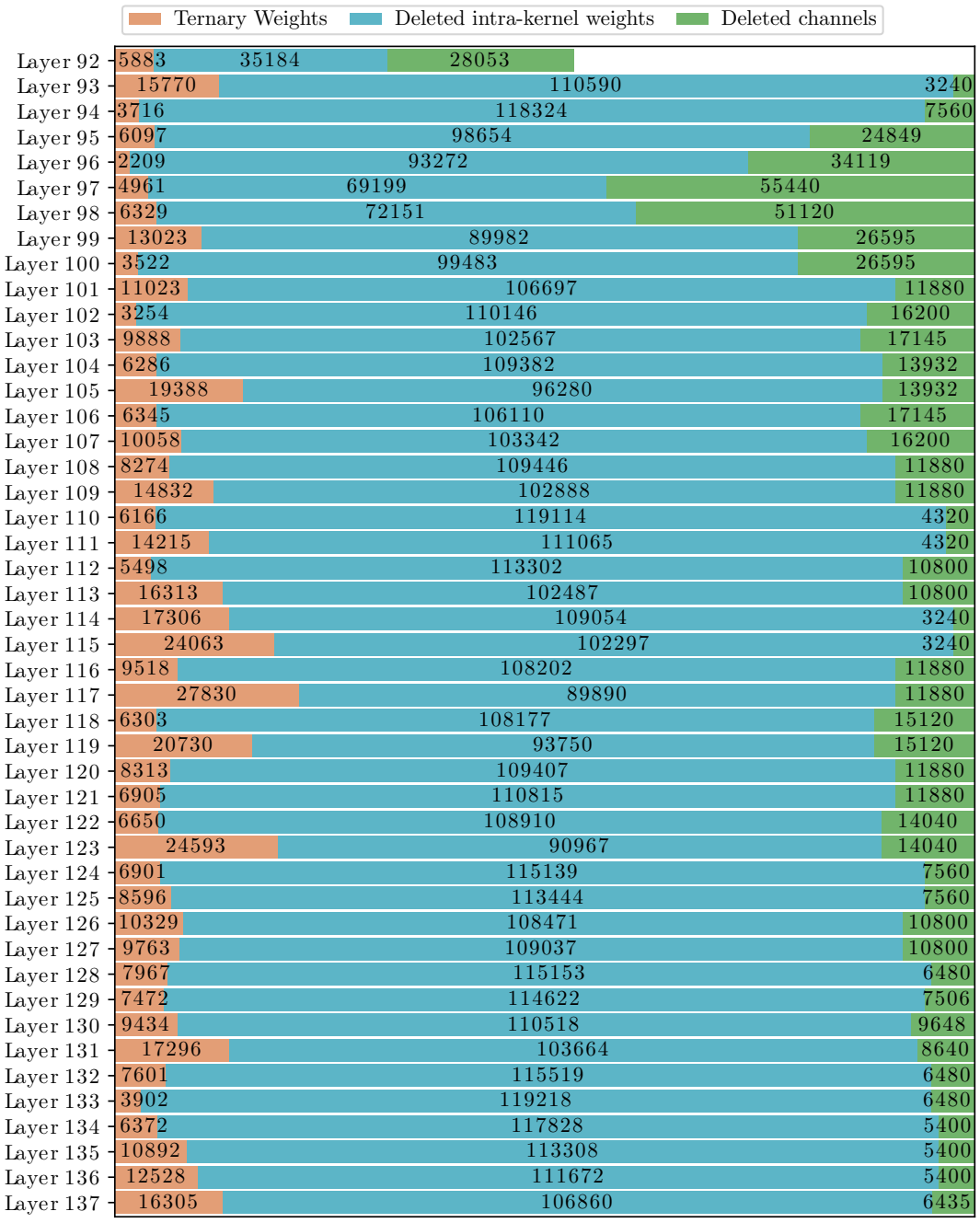

FIGURE 8.3: Per layer sparsity and number of deleted weights due to implicit channel pruning of C100-MicroNet stage three after applying EC2T.