Pruning and Quantization for Efficient Deep Neural Networks

Lukas Enderich

Technische Fakultät
Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

Betreuer: Prof. Dr. Wolfram Burgard

Pruning and Quantization for Efficient Deep Neural Networks

Lukas Enderich

Dissertation zur Erlangung des akademischen Grades Doktor der Naturwissenschaften
Technische Fakultät, Albert-Ludwigs-Universität Freiburg

Dekan	Prof. Dr. Rolf Backofen
Erstgutachter	Prof. Dr. Wolfram Burgard Albert-Ludwigs-Universität Freiburg
Zweitgutachter	Prof. Dr. Bin Yang Universität Stuttgart
Tag der Disputation	24.02.2021

Zusammenfassung

In den letzten Jahren haben tiefe neuronale Netze bewiesen, dass sie klassische Methoden bei verschiedenen Aufgaben des maschinellen Lernens übertreffen. Solche künstlichen neuronalen Netze machen Vorhersagen auf Basis von Mustervergleichen und werden anhand von Erfahrungenswerten trainiert. Durch die Nutzung großer Datenmengen sind sie in der Lage, hierarchische Repräsentationen von rohen Eingabedaten zu lernen und somit das Lernen von Merkmalen sowie die Klassifikation zu kombinieren. Allerdings sind eine hohe anfängliche Modellkapazität sowie Fließkommaoperationen erforderlich, um ein tiefes neuronales Netzwerk erfolgreich von Grund auf zu trainieren. Daher sind die trainierten Modelle meist überparametrisiert und benötigen leistungsfähige Recheneinheiten.

Im Gegensatz dazu verfügen mobile Endgeräte nur über begrenzte Ressourcen in Bezug auf Speicher-, Energie-, und Rechenkapazität. Dadurch wird die Komplexität von neuronalen Netzen stark begrenzt. Für den Gebrauch auf Geräten mit begrenzter Kapazität werden daher Reduktionsmethoden eingesetzt, um die Komplexität der trainierten Modelle zu verringern. Einerseits reduzieren Quantisierungsmethoden die Bitgrößen von Operanden und Operationen, wodurch der Speicherbedarf unmittelbar gesenkt wird. Darüber hinaus reduzieren Quantisierungen mittels Festkommaarithmetik zusätzlich den Rechen- und Energieaufwand auf dedizierter Hardware. Andererseits wird beim sogenannten Pruning die Anzahl der Operanden und Operationen verringert, indem redundante Netzwerkverbindungen gelöscht werden. Hier reduziert das Löschen ganzer Filter und Neuronen (genannt Filter Pruning) direkt die Speicher-, Energie- und Berechnungskomplexität, ohne dass spezielle Hardware erforderlich ist. Ein gängiger Ansatz ist daher, zunächst ein großes und überparametrisiertes Modell zu trainieren und es im Anschluss mit geeigneten Reduktionstechniken zu komprimieren.

Allerdings gibt es mehrere Probleme mit bisherigen Reduktionsansätzen. Zum einen sind viele von ihnen aufwendig zu implementieren oder müssen außerhalb der für neuronale Netze üblichen Optimierungskette gelöst werden. Außerdem ist die Vernachlässigung von Bedingungen für Festkommazahlen ein häufiges Problem bei Quantisierungsmethoden. Zum anderen ist es meist nicht möglich, die entscheidenden Einschränkungen der Zielhardware exakt zu spezifizieren, was iterative Reduktionsverfahren notwendig macht.

In der zugrundeliegenden Arbeit adressieren wir diese Probleme mit verschiedenen Ansätzen zu Pruning und Quantisierung. Unsere Ansätze bestehen jeweils aus einem Reduktionsfehler, der mit geringem Implementierungsaufwand in das übliche Trainingsverfahren für neuronalen Netzen integriert werden kann. Die Minimierung dieses Reduktionsfeh-

lers während des Trainings reduziert die Modellkomplexität entweder durch Quantisieren mittels Festkommaarithmetik, durch Filter Pruning, oder durch eine Kombination aus beidem.

Zunächst stellen wir einen einfachen und effizienten Reduktionsfehler vor, um tiefe neuronale Netze mit multimodalen Gewichtsverteilungen und minimalem Quantisierungsfehler zu trainieren. Dadurch können die Gewichte nach dem Training ohne signifikanten Genauigkeitsverlust in eine Darstellungen mittels Festkommaarithmetik überführt werden. Somit is unser Ansatz sehr einfach zu implementieren und erbringt selbst für sehr kleine Bitgrößen eine hervorragende Leistung. Darüber hinaus erweitern wir unseren Ansatz, indem wir sowohl die Batch-Normalisierungsschichten als auch die Aktivierungsfunktionen berücksichtigen. Auf diese Weise ist es möglich, tiefe neuronale Netze zu trainieren, die im Anschluss an das Training ohne Fließkommaoperationen ausgewertet werden können.

Als Nächstes schlagen wir eine neue Filter Pruning Methode vor, die in der Lage ist, die Anzahl der Parameter und Multiplikationen eines tiefen neuronalen Netzwerks basierend auf einer bestimmten Zielgröße zu reduzieren. Diesbezüglich ist der Nutzer in der Lage, Maximalwerte für die Anzahl an Parametern und Multiplikationen entsprechend den Speicher- und Rechenressourcen des Zielgeräts zu definieren. Während des Trainings berechnet ein Reduktionsfehler die Differenz zwischen der tatsächlichen Modellgröße und der Zielgröße in Bezug auf die Anzahl der Parameter und Multiplikationen. Dieser Reduktionsfehler wird minimiert, indem ganze Filter und Neuronen über die Kanäle der Batch-Normalisierungsschichten gelöscht werden. So kann eine globale Lösung gefunden werden, die die Bedingungen der Zielhardware erfüllt und zugleich möglichst viel Leistung in Bezug auf die Lernaufgabe erhält.

Schließlich schlagen wir eine neue und besonders effiziente Kombination aus Filter Pruning und Quantisierung mittels Festkommaarithmetik vor. Hier definieren wir Komplexität zunächst als ein Zusammenschluss aus vier essentiellen Metriken: den Speicherbedarf, die aus der Anzahl der Bitoperationen resultierende Rechenkomplexität, die aus der Kommunikation zwischen der Recheneinheit und dem Speicher resultierende Bandbreite, und die maximalen Speicherkosten der Aktivierungen. Basierend auf diesen vier Metriken berechnet der Reduktionsfehler nach jedem Trainingsschritt die Differenz zwischen der tatsächlichen Modellkomplexität und den auf dem Zielgerät verfügbaren Ressourcen. Durch den Einsatz von speziell entwickelten Pruning- und Quantisierungsmodulen kann der Reduktionsfehler während des Training durch eine Kombination aus Festkomma-Quantisierung und Filter Pruning minimiert werden. Das trainierte Modell ist somit hocheffizient: Es hat keine Batch-Normalisierungsschichten, rechnet alle Parameter und Aktivierungen in Festkommaarithmetik, und erfüllt die Komplexitätsmetriken des Zielgerätes.

Abstract

In recent years, deep neural networks have proven to outperform classical methods on several machine learning tasks. Such deep networks make predictions based on pattern matching and receive training based on experience. By leveraging large amounts of data, they are capable of learning hierarchical representations of raw input data and thus combine both feature learning and classification. However, a high initial model capacity as well as floating-point operations are required to successfully train a deep neural network from scratch. As a result, trained models are usually over-parameterized after training and require powerful processing units.

In contrast, both mobile and embedded devices have finite resources regarding their memory, energy, and computational capacity. This severely limits the complexity of neural networks. Nevertheless, to make them available for use on devices with limited capacity, reduction methods are employed to reduce the complexity of trained models. On the one hand, quantization methods reduce the bit sizes of operands and operations, which immediately decreases the memory requirements. Furthermore, fixed-point quantization methods additionally reduce the computational and energy requirements on dedicated hardware. On the other hand, pruning reduces the number of operands and operations by removing redundant network connections. Furthermore, pruning entire filters and neurons from the network architecture directly reduces the memory, energy, and computational complexity without the need for specialized hardware. A common approach is therefore to first train a large and over-parameterized network and then reduce it using appropriate reduction methods.

However, there are several problems with previous approaches. First, many of them are either complicated to implement or must be solved outside the standard optimization procedure of deep neural networks. Moreover, neglecting fixed-point constraints is a common problem with quantization approaches. On the other hand, it is usually not possible to directly specify the critical limitations of the target hardware, which makes iterative reduction procedures necessary.

In the underlying thesis, we address these problems by different contributions to both pruning and quantization. Each of our approaches consists of a reduction loss that can be integrated into the common training procedure of deep neural networks with little implementation effort. Minimizing the reduction loss during training reduces the model complexity either by fixed-point quantization, filter pruning, or a combination of both.

At first, we propose a simple and efficient reduction loss to train deep neural networks

with multi-modal weight distributions and minimal quantization error. Consequently, the weights can be quantized into fixed-point representations after training with no significant loss in accuracy. Thus, we present an approach that is very easy to implement and yields excellent performance even for small bit sizes. Furthermore, we extend our approach by taking into account both the batch-normalization layers and activation functions. In this way, it is possible to train deep neural networks that can be evaluated without floating-point operations after training.

Next, we propose a novel filter pruning method that is capable of reducing the number of parameters and multiplication of a deep neural network based on a given target size. Therefore, the user is able to define maximum values for both the number of parameters and multiplications according to the memory and computational resources of the target device. During training, the reduction loss calculates the difference between the actual model size and the target size in terms of the number of parameters and required multiplications. Furthermore, the reduction loss is minimized by pruning whole filters and neurons via the channel-wise affine transformation of the batch-normalization layers. In this way, a global selection of filters and neurons can be found that, on the one hand, solves the learning task in the best possible way and, on the other hand, fulfills the constraints of the target device.

Finally, we propose a novel and highly efficient combination of filter pruning and fixed-point quantization. Here, we define complexity as an aggregation of four essential metrics: the memory requirement, the computational complexity resulting from the number of bit operations, the bandwidth resulting from the communication between the processing unit and the memory, and the maximum storage cost of the activations. Based on these four metrics, the reduction loss calculates the difference between the actual model complexity and the resources available on the target device. The reduction loss can be minimized during training by using pruning and quantization layers specially developed by us for this purpose. The trained model is thus highly efficient: it runs without batch-normalization layers, has all parameters and activations in fixed-point representation, and fulfills the complexity metrics of the target device.

Acknowledgments

Now I would like to thank all the great people who stood by me during the last years, who gave me support and strength, and with whom it was always a pleasure to work.

First of all, I would like to thank Wolfram Burgard for allowing me to write my thesis under his excellent guidance. He gave me a lot of confidence right from the start and always provided me with honest advice. Since I was an external candidate who had only a limited time of three and a half years to complete his PhD, this was anything but a matter of course, and I am very grateful for doing so. I highly value his competence, both professional and personal.

Besides, I want to thank Bin Yang for being such a good professor and giving such good lectures. I visited nearly all of his readings during my master's studies, which also introduced me to the concept of machine learning for the first time. In the following, I wrote both my student research project and my master's thesis at his institute. Without such good preparation, I am sure I would not have been able to complete my PhD over the last three and a half years.

Furthermore, I would like to thank Bosch and all its employees for allowing me to write my thesis in such an innovative company and with such a good employer. At this point, special thanks go to Andre Treptow and Lothar Baum. Andre Treptow was already my group leader during my time as an intern and later as a working student. Even then he allowed me to contribute my ideas and to work on important and exciting future technologies. Together with Lothar Baum, he supported my wish to do my PhD at Bosch and cleared all hurdles. It was Lothar Baum who provided the important and valuable contact with Wolfram Burgard for the first time.

However, very special thanks go to Fabian Timm, my mentor throughout my time at Bosch. He hired me as an intern, kept me as a working student during my master's thesis, and finally took me on as a doctoral student after my studies. During these five years, I learned quite a lot from him, both professionally and personally. He always stood by me, even during difficult times. He also gave me a lot of technical support, and we wrote several papers together.

In addition, I would like to thank my wonderful colleagues Jasmin Ebert, Julia Lust, Matthias Rath, and Jonathan Stysch. All of them supported me during several paper deadlines, gave me great advice, and proofread a lot of my work. We also enjoyed great moments together in our free time and became good friends.

Last but not least, I would like to thank my family and my friends for believing in me

and always being around for me. I would not have been able to write this thesis without you.

Contents

Chapter 1

Introduction

Thinking, Fast and Slow is the title of a bestseller by Daniel Kahneman published in 2011 [1]. Therein Kahneman summarizes his scientific work on cognitive psychology and decision making and introduces his thesis that the way of thinking can be divided into two categories. The first way of thinking (*System 1*) is fast, intuitive, error-prone, and especially effective in solving everyday decisions [1]. It is directly related to our emotional, visual, and acoustic perception system. In contrast, the second way of thinking (*System 2*) is slow, conscious, thoughtful, and reliable in making complex decisions [1]. *System 2* has the ability to reason and to control its attention towards specific objects or circumstances. Furthermore, it contains a physical model of the world that surrounds it and can thus verify its own decisions.

Considering the current research in the field of Machine Learning, it seems that the progress in Deep Learning resembles a *System 1* rather than a *System 2*. Deep Learning describes an information processing system based on deep neural networks [2, 3]. Such deep neural networks make predictions based on pattern matching, receive training based on experience, and automatically learn hierarchical representations of raw input data. However, deep neural networks have not yet been able to reason nor to represent causality within their architecture [4]. Verifying the predictions made is one of the major open problems in Deep Learning [4].

However, taking into account the importance of *System 1* for solving everyday problems, the potential benefit of Deep Learning for people's lives becomes visible. Even today, deep neural networks solve challenging tasks in computer vision, object detection, and speech recognition [2, 5, 6]. In private transport, driver assistance systems increasingly utilize deep learning algorithms to improve both driving comfort and safety [7]. Future highway pilots will combine algorithms of classical robotics and machine learning systems including deep learning to perform localization, perception, and navigation. Furthermore, medical research is increasingly relying on data-driven approaches [8]. For example, many of the vaccines against Covid-19 have been developed by means of artificial intelligence and data-driven approaches [9].

Nevertheless, there are still a number of problems regarding the application of deep neural networks. Besides their inability to represent causality, the performance of deep

neural networks depends on both their initial model size and the amount of data used for training [10]. As a matter of fact, the greatest results have been accomplished by training large models with many parameters using large amounts of training data [10, 11]. In this context, Frankle and Corbin showed the correlation between the initial model size and the probability of effective training: The higher the number of parameters that are initialized with random values, the greater the chance of initializing a subset of the parameters with values that represent an effective baseline for training (Frankle and Corbin refer to this subgraph as the *Lottery Ticket* [10]). As a result, deep neural networks are usually over-parameterized after the training and have high memory requirements. Furthermore, floating-point multiplications are especially expensive in terms of computation time and energy consumption [12, 13].

Due to massive GPU parallelization, the computational effort required to train deep neural networks is no longer a critical bottleneck. In contrast, mobile devices, as well as embedded systems, have a very limited memory capacity, may only consume little energy, and have significantly fewer parallelization capabilities [12]. This leads to two major problems. First, starting with a significantly reduced model capacity usually downgrades the training progress of deep neural networks [10] (see the description of the *Lottery-Ticket* in the previous paragraph). Second, distributing the limited capacity across the individual layers is a challenging problem: finding both a suitable network depth and layer widths requires a time-consuming hyper-parameter search.

As a consequence, reduction techniques have been developed to reduce the complexity of trained and over-parameterized deep neural networks [14, 15, 16, 17, 18]. Approaches that are capable of reducing both the memory requirements and the computational complexity can be grouped into two categories [12]. On the one hand, quantization reduces the precision of the parameters and activations by reducing their respective bit size. On the other hand, pruning, factorization, and network distillation reduce the number of parameters and activations by removing redundant network connections. While pruning sets single weights our groups of weights to zero, factorization decomposes weight matrices into the product of two low-rank matrices. Furthermore, network distillation utilizes an over-parameterized teacher network to train a smaller student network by using both the soft targets of the trained teacher and the known class labels for training [19]. An overview is given in Figure 1.1 and can also be found in [12]. In this thesis, we make contributions to network quantization, network pruning, and its joint combination.

Network quantization is achieved by having several weights sharing the same value. Although unconditional weight sharing (which is also called weight clustering) can significantly reduce memory costs [20], additional constraints are needed to enable its benefits on various devices. Arbitrary weight clusters require lookup tables during each forward pass and thus lead to cumbersome data accesses. In contrast, fixed-point representations of the weights and activations enable both reduced memory requirements and efficient processing on dedicated hardware [12, 13]. Fixed-point numbers consist of an integer

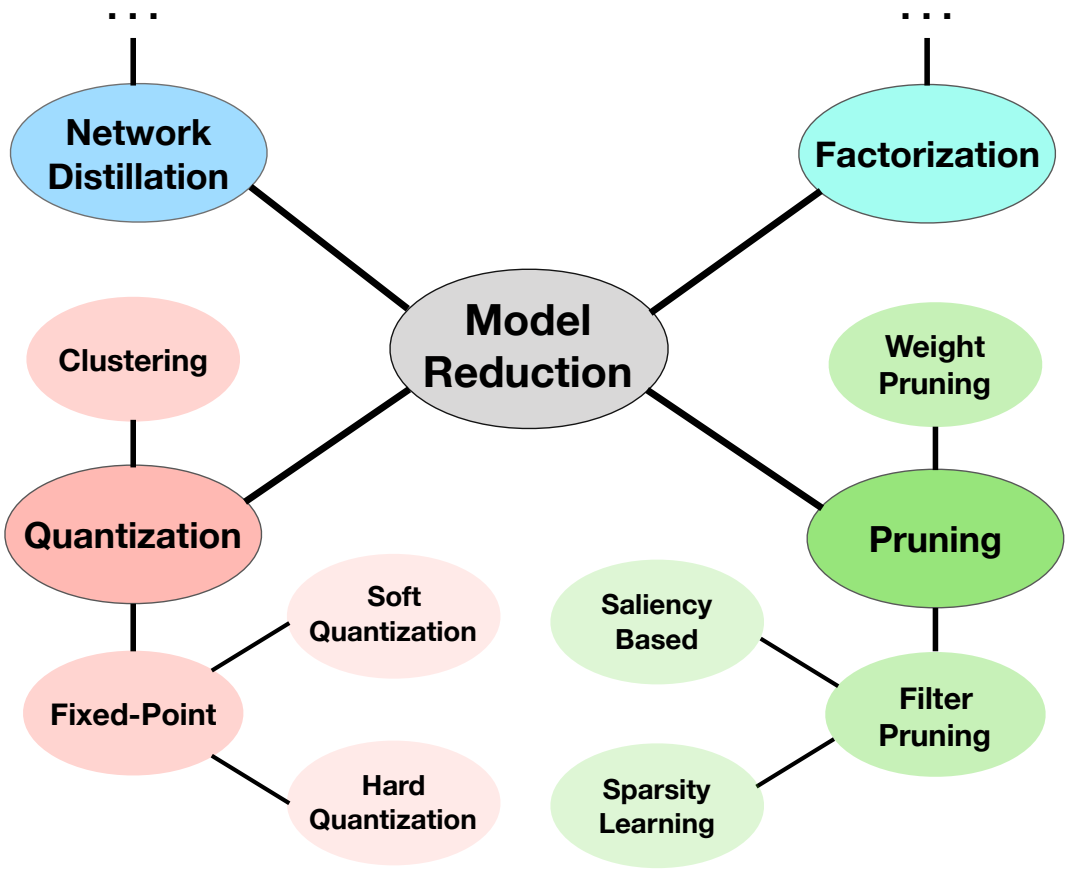

Figure 1.1: An overview of different approaches that are capable of reducing the complexity of deep neural networks. In general, model reduction techniques can be divided into four subgroups. On the one hand, pruning, factorization, and network distillation aim at reducing the number of parameters and multiplications of deep neural networks [12]. Here, pruning describes the process of deleting redundant network connections by setting weight values to zero. This can either be done by pruning single weights (i.e. weight pruning) or complete filters and neurons (i.e. filter pruning). In contrast, factorization reduces the tensor sizes of weights and activations by decomposing the weight-tensor or -matrix of single layers. Furthermore, network distillation transfers the knowledge learned by a complex teacher network to a smaller student network. This is done by utilizing both the known hard labels of the classification task and the soft targets predicted by the teacher network. On the other hand, quantization reduces the precision (i.e. the bit sizes) of weights and activations, which immediately reduces the memory requirements of the network [12]. Furthermore, if both the weights and activations are quantized using fixed-point arithmetic, the computational effort that is needed to evaluate the model can be significantly reduced on dedicated fixed-point hardware. In this thesis, we make contributions to the field of fixed-point quantization and filter pruning. Furthermore, we provide an efficient combination of both approaches.

and a power-of-two scaling factor that indicates the position of the decimal point [12]. Using the same scaling factor for all weights of a given group (e.g. for all weights of one network layer) can significantly increase the efficiency of data processing. In Chapter 4, we propose a novel reduction loss to train deep neural networks with minimal quantization error. During training, the distribution of the weights changes from a uni-modal distribution to a symmetric and multi-modal distribution, where each mode belongs to a particular fixed-point number. After training, the network weights can be converted into a fixed-point representation with no significant loss in accuracy. The proposed method is a soft quantization approach, which means that all parameters remain in floating-point precision during training and are only quantized afterwards. In addition, we integrate the batch-normalization layers into the reduction loss of our soft quantization approach and replace the ReLU activation functions with an unsigned fixed-point quantization function. Thus, we can train deep neural networks that can be evaluated without the use of floating-point operations after training.

Network pruning is achieved by inducing sparsity constraints among network connections. Such sparsity constraints can either be unstructured or structured [21]. Setting individual weight values to zero leads to unstructured weight tensors or matrices. During inference, multiply-and-accumulate operations that access zero weights do not have to be computed [12]. However, unstructured sparsity does not change the tensor sizes of weights and activations and therefore has only little impact on the overall memory requirements of deep neural networks. In contrast, filter pruning (which is also called channel pruning) deletes entire filters and neurons from the network architecture and thus directly decreases the tensor sizes of both the weights and activations [22, 23, 24]. As a result, filter pruning reduces the number of parameters and required multiplications without the need for specialized hardware. In Chapter 5, we propose a novel filter pruning method to reduce the complexity of deep neural networks based on a given target size that is specified by the user according to the number of parameters and multiplications that are available on the target device. During training, a reduction loss is minimized that calculates the difference between the desired and the actual model size. Thus, the algorithm automatically distributes the pruning budget across the individual layers such that the target size is fulfilled. Here, filter pruning is induced via the affine transformations of the batch-normalization layers, so no additional variables are needed.

Combined pruning and quantization reduces the total number of parameters and activations as well as their respective bit sizes. Furthermore, a combination of fixed-point quantization and filter pruning is especially effective in reducing both the memory and computational effort of deep neural networks. However, bringing the two approaches together in one optimization problem is complex and most of the proposed concepts apply pruning and quantization separately even though both influence each other [20, 25, 26]. For example, there are infinite combinations of possible pruning and quantization rates to halve the memory requirements of a single layer. The problem becomes even more complex:

when reducing a multi-layer neural network, each layer can have different bit sizes and pruning rates to fulfill the overall target size. In Chapter 6, we propose a holistic approach for reducing the complexity of deep neural networks by simultaneous filter pruning and fixed-point quantization. First, we define complexity using four essential metrics that directly result from the network architecture: the memory requirement, the computational effort during inference, the bandwidth, and the maximum storage effort required for the activations. By defining maximum values for each of these metrics, the user specifies a comprehensive target size. Based on this, we propose a computational graph equipped with custom layers that allow to change the architecture during training and consequently to reduce complexity. Thus, we achieve high reduction rates with at the same time excellent performance on several network architectures.

1.1 Scientific Contributions

We contribute to the field of model reduction by various approaches to quantization, pruning, and a combination of both. In general, our approaches include a reduction loss $\mathcal{L}_{\text{reduce}}$ whose minimization leads to a reduction of the model complexity. The reduction loss can be integrated into the common training procedure of deep neural networks by combining it with the learning loss $\mathcal{L}_{\text{learn}}$. Consequently, the overall training objective is composed as follows:

$$\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda \, \mathcal{L}_{\text{reduce}} \, . \tag{1.1}$$

Here, λ is the regularization parameter that controls the weighting between the learning loss and the reduction loss. In this way, both can be minimized simultaneously by gradient descent. In detail, we make the following contributions:

- In Chapter 4, we propose a novel soft quantization approach for deep neural networks with fixed-point weights. We introduce a reduction loss whose minimization during training causes the distribution of weights to change from a uni-modal to a multi-modal distribution. Thus, the training is done in floating-point precision and simultaneously reduces the expected quantization error. Our approach has comparatively low implementation effort and delivers excellent performance, especially for low bit sizes. Furthermore, we amplify our approach and propose the first soft quantization that takes into account the parameters of the batch-normalization layers. By quantizing the layer activations as well, the trained networks can be evaluated using pure fixed-point arithmetic.

- In Chapter 5, we propose a novel filter pruning method to reduce the complexity of deep neural networks based on a given target size. Here, the user can define the target size by the number of parameters and multiplications that are available on

the target device. During training, the reduction loss that indicates the difference between the actual model size and the target size is minimized by pruning complete filters and neurons from the network architecture. By allowing each layer to have its own pruning rate, the algorithm distributes the pruning budget across the individual layers such that a global solution is found.

- In Chapter 6, we propose the first combination of filter pruning and fixed-point quantization that reduces the complexity of trained networks based on four essential metrics: the memory requirement, the computation effort during inference, the bandwidth, and the maximum storage effort required for the activations. The user is able to define maximum values for each of these metrics, with the reduction loss indicating the deviation between the actual model complexity and the target complexity based on these four metrics. By learning individual layer widths and bit sizes, the resources available are distributed automatically such that the target complexity is not exceeded.

1.2 Publications

The work presented in this thesis is based on the following publications in conference proceedings and journals:

- Lukas Enderich, Fabian Timm, Lars Rosenbaum, and Wolfram Burgard. Learning Multimodal Fixed-Point Weights using Gradient Descent. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, Bruges, Belgium, 2019.

- Lukas Enderich, Fabian Timm, and Wolfram Burgard. SYMOG: Learning symmetric mixture of Gaussian modes for improved fixed-point quantization. In *Proceedings of the Neurocomputing Journal*. Year 2020, Volume 4/6, Pages 310 - 315.

- Lukas Enderich, Fabian Timm, and Wolfram Burgard. Holistic Filter Pruning for Efficient Deep Neural Networks. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2021.

Chapter 2

Background Theory

This chapter briefly introduces the theoretical concepts and definitions on which the contributions of this thesis are built. First, Section 2.1 presents the theoretical foundations of deep neural networks including the different layer types, loss functions, and the optimization procedure, which is based on the backpropagation algorithm. The mathematical interpretation of a deep neural network being a parametric function approximation used in this thesis derives primarily from Goodfellow *et al.* [3]. Second, Section 2.2 introduces the numerical representations of fixed-point numbers and their advantages over floating-point representations. Based on this, Section 2.3 introduces the basic concepts of fixed-point quantization functions. Here, we also give an example to illustrate the computational path of a neural network layer that is converted into a fixed-point representation.

2.1 Deep Neural Networks

Deep neural networks form a special class of computational graphs parameterized by a set of learnable parameters $\Theta = \{w_1, ..., w_M\}$. Here, M is the number of parameters, and $w \in \Theta$ are called weights. Thus, a neural network is a parameterized function $f_\Theta : \mathbb{R}^N \rightarrow \mathbb{R}^C$ which maps an input signal $x \in \mathbb{R}^N$ to an output signal $\widehat{y} \in \mathbb{R}^C$, where N represents the input dimension and C the output dimension. In a Deep Learning system, $\widehat{y}$ is usually referred to as the prediction of a certain learning task, which is in turn described by the unknown target function $f^* : \mathbb{R}^N \rightarrow \mathbb{R}^C$. Here, f^* maps the same input signal $x \in \mathbb{R}^N$ to $y \in \mathbb{R}^C$, the corresponding correct output of the learning task, which may represent a discrete class label (if the learning task is a classification problem with C classes) or a continuous variable (if the learning task is a regression problem of dimension C).

In supervised learning, the parameters of a neural network can be adjusted by observing a set of training examples $D = \{(x_i, y_i)\}_{i=1}^d$. Here, y_i is the known output of the corresponding input example x_i, and d indicates the number of training examples. The training seeks to adjust the parameters of the network such that the trained model is capable of making correct predictions on previously unseen data. Thus, observing empirical data is used for approximating the unknown target function f^* of the learning task by adjusting the set of

parameters Θ accordingly. In general, the training consists of several epochs, during each of which the set of training examples is observed once. The whole procedure consists of six basic steps, parts of which are repeated in sequential order:

1. **Initialization**: The parameters of the network are initialized with random values.

2. **Shuffling**: Before each epoch, the training data is randomly shuffled and divided into so-called mini-batches of certain batch size. These mini-batches are successively propagated through the network and utilized to adjust its parameters.

3. **Forward pass**: A mini-batch of training examples x is propagated through the network to calculate the predicted output according to $\widehat{y} = f_\Theta(x)$.

4. **Loss calculation**: A loss functions $\mathcal{L}\left(\widehat{y}, y\right)$ calculates the difference between the network prediction $\widehat{y}$ and the correct output y of the current mini-batch.

5. **Backward pass**: The loss is derived with respect to the predicted output $\widehat{y}$. The resulting gradient is propagated backward through the network to calculate the gradients for the network parameters $w \in \Theta$. This is done by applying the chain rule at each node of the computational graph.

6. **Update step**: After calculating the gradients with respect to the parameters, each parameter is updated in the direction of its negative gradient:

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w} . \tag{2.1}$$

   Here, the increment of the update step is defined by the learning rate η. There are several strategies for improving the adaptation of the weights including momentum, Nesterov momentum, running averages, and learning rate schedules [27, 28]. Widely used optimizers are Stochastic Gradient Descent (SGD) [28] and Adam [27].

Following the update step, the training returns to the next forward pass (see step 3.). After a single epoch is completed (meaning that the training data has been observed once), the training returns to the shuffling process (see step 2.). This continues until either the desired maximum number of iterations is reached, or a termination criterion is fulfilled.

A deep neural network consists of one input layer, at least one hidden layer, and one output layer. Different types of layers with different functionalities are available for this purpose. These layer types are described in the following section. Subsequently, widely used loss functions as well as the basic concept of the backpropagation algorithm are introduced.

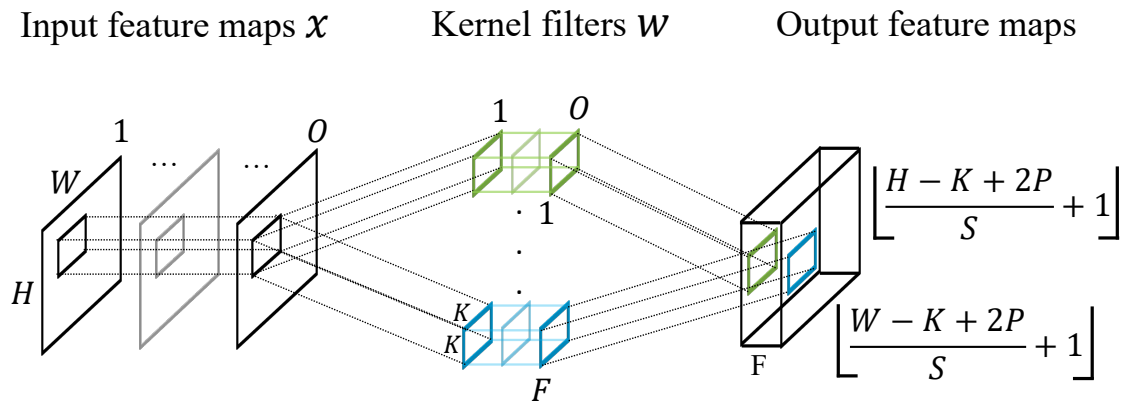

Figure 2.1: A simplified representation of a convolutional layer. The input consists of O feature maps of size $W \times H$. The weights consist of F kernel filters of size $K \times K \times O$. The filters are convoluted over the width and height of the input tensor, each of which computing a two-dimensional output feature map. The size of these output feature maps depends on the input and filter dimensions, the stride S, and the number of zeros P padded to each spatial input dimension.

2.1.1 Layer Types

A feed-forward deep neural network consists of several sequentially stacked layers, each layer operating on the output of the previous layer:

$$\widehat{y} = f_\Theta(x) = f^L\left(\ldots f^2\left(f^1\left(x,\, w_1\right),\, w_2\right),\, \ldots w_L\right)\,. \tag{2.2}$$

Here, $l \in \{1, \ldots, L\}$ is the layer index with L being the number of layers, f^l is the layer of index l, w_l denotes the weights used in layer l, and $\Theta = \{w_1, ..., w_L\}$ is the set of learnable parameters. Different types of layers perform, for instance, weighted sums, batch-normalization, pooling, or non-linear transformations. The following is a brief description of the layer types that are used in the experimental setup of this thesis.

Fully-connected Layers: A fully-connected layer is a single-layer perceptron that consists of one or several neurons. Each neuron is connected to all input values x_i of the input vector $x \in \mathbb{R}^N$ and computes the following weighted sum:

$$a = \sum_{i=1}^{N} w_i\, x_i + b\,. \tag{2.3}$$

Here, N is the number of input values, w_i the corresponding weight value, b the added bias, and a the weighted sum which is calculated by the neuron. Furthermore, if the fully-connected layer consists of several neurons, the output is a one-dimensional vector containing the weighted sums of each neuron. With $x_{l-1} \in \mathbb{R}^N$ being the input of layer l, the output can be calculated by:

$$a_l = w_l\, x_{l-1} + b_l\,. \tag{2.4}$$

Here, $w_l \in \mathbb{R}^{M \times N}$ is the weight matrix of layer l, $b_l \in \mathbb{R}^M$ the corresponding bias vector, M the number of neurons, and $a_l \in \mathbb{R}^M$ the layer output. Both the weights and the bias can be adjusted during training to adapt the linear transformation of the layer according to the training data. Since each output neuron interacts with each input neuron, the learned features span the entire visual field and thus possess global properties. Consequently, fully-connected layers are usually located at the end of the network to perform the classification.

Convolutional Layers: Unlike fully-connected layers, convolutional layers perform the convolution operator instead of ordinary matrix multiplications. More precisely, each layer consists of one or several filters, each performing a discrete convolution over a multi-dimensional input space. The filters contain learnable parameters, have certain spatial dimensions but cover the entire depth of the input signal. Taking into account the stride, each filter is convoluted over the width and height of the input tensor to compute a two-dimensional feature map. Hence, each feature map contains weighted sums calculated from the same filter weights but different input values. Therefore, convolutional layers possess significantly fewer parameters than fully-connected layers and compute local features that contain spatial information. The number of output feature maps is equal to the number of filters used.

Considering a 2D convolutional layer as shown in Figure 2.1, $x \in \mathbb{R}^{W \times H \times O}$ represents the layer input consisting of O feature maps of size $W \times H$. The filter kernel $w \in \mathbb{R}^{K \times K \times O \times F}$ consists of F filters of size $K \times K \times O$ with K being the kernel size. In most cases, quadratic filters are used to calculate the convolution. However, it is also feasible to use non-quadratic filters with different spatial dimensions. Furthermore, the input tensor can optionally be padded with zero values to preserve its spatial dimensions (zero-padding). Hence, the layer output consists of F feature maps of size

$$\left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1 \times \left\lfloor \frac{H - K + 2P}{S} \right\rfloor + 1 \,. \tag{2.5}$$

Here, $\lfloor \cdot \rfloor$ rounds to the subjacent integer, S indicates the stride by which the filters are convoluted over the input signal, and P is the number of zeros padded to each input dimension. After the convolution, a bias $b \in \mathbb{R}^F$ is optionally added to the F output feature maps.

Batch-normalization Layers: Both fully-connected and convolutional layers calculate weighted sums. With l being the layer index and x_{l-1} being the input of layer l, the computation of both layers can be summarized as

$$a_l = w_l * x_{l-1} + b_l \,. \tag{2.6}$$

Here, $*$ is either a convolution operator or a matrix-vector multiplication, w_l is either a weight tensor or matrix, and b_l is the bias vector. The layer output a_l consists of several

channels, the number of channels being equal to the number of convolution filters or matrix rows in w_l. Following the weighted sums computed by either a convolutional or a fully-connected layer, batch-normalization layers are frequently used to first normalize and then linearly transform each channel of a_l. Therefore, the output $\widehat{a}_c$ of the batch-normalization layer can be written as

$$\widehat{a}_{l,c} = \begin{cases} \dfrac{a_{l,c} - \mathrm{E}[a_{l,c}]}{\sqrt{\mathrm{Var}[a_{l,c}] + \epsilon}} \gamma_{l,c} + \beta_{l,c} & \text{during training,} \\ \dfrac{a_{l,c} - \mu_{l,c}}{\sqrt{\sigma^2_{l,c} + \epsilon}} \gamma_{l,c} + \beta_{l,c} & \text{during inference,} \end{cases} \tag{2.7}$$

where c is the channel index, $\mathrm{E}[a_c]$ and $\mathrm{Var}[a_c]$ are the mean and variance of the current mini-batch (the so-called batch statistics), μ_c and σ_c are the running estimates of the mean and the variance, and $\{\gamma_c, \beta_c\}$ are the learnable parameters of the linear transformation. During the training, the normalization is done using the batch statistics, whereas the running estimates of the mean and the variance are continuously updated. During the inference, the running estimates are used for normalization.

After the training, batch-normalization layers can be folded into the preceding convolutional or fully-connected layers to accelerate the forward pass and reduce the memory requirements [13]. Combining Equation 2.6 and Equation 2.7, the calculation of the folded layers during the inference can be written as

$$\widehat{a}_l = \widehat{w}_l * x_{l-1} + \widehat{b}_l \tag{2.8}$$

$$\text{with} \quad \widehat{w}_l = w_l \frac{\gamma_l}{\sqrt{\sigma^2_l + \epsilon}} \quad \text{and} \quad \widehat{b}_l = (b_l - \mu_l) \frac{\gamma_l}{\sqrt{\sigma^2_l + \epsilon}} + \beta_l \,. \tag{2.9}$$

Here, $\widehat{w}$ and $\widehat{b}$ are the parameters of the folded layers.

Non-linear Activation Functions: In order to enable non-linear transformations, deep neural networks use non-linear activation functions subsequent to batch-normalization layers. For example, if $\widehat{a}_l$ is the output of the batch-normalization layer with index l, the non-linear activation function calculates the layer activation as follows

$$x_l = \sigma\left(\widehat{a}_l\right) \,. \tag{2.10}$$

The most common non-linear activation function is the rectified linear unit (ReLU [29]) that sets all negative input values to zero. The computation can therefore be written as

$$x = \mathrm{ReLU}\left(\widehat{a}\right) = \begin{cases} \widehat{a} & \text{if } \widehat{a} \geq 0 \\ 0 & \text{if } \widehat{a} < 0 \end{cases} \,. \tag{2.11}$$

Whereas the ReLU activation function is commonly used between hidden layers to extract meaningful features, the softmax activation function is a widely used output for classification problems. The softmax function maps an input vector $x \in \mathbb{R}^N$ to an output of the same size, with each value being restricted to the domain $(0, 1]$ (hence: $\text{softmax}(x_i) \in (0, 1]$). Furthermore, the sum of all output values accumulates to one (hence: $\sum_{i=1}^{N} \text{softmax}(x_i) = 1$). The computation is as follows

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{N} \exp(x_j)} \,. \tag{2.12}$$

Due to its range of values and accumulative property, the softmax output is widely used for representing class probabilities in a classification problem. Thus, it usually forms the output layer, with each output value representing a single class probability.

Other commonly used non-linear activation functions are sigmoid, hyperbolic tangent, or Leaky ReLU [3].

Pooling Layers: Pooling layers reduce the spatial dimensions of their input feature maps. They are usually located after a convolutional layer or a batch-normalization layer and offer different pooling operations. For example, the spatial dimension can be reduced by selecting only the maximum of a two-dimensional group of input values (max-pooling). Other pooling operations can for example be based on the average calculation (average-pooling), the L^2-norm, or on stochastic pooling [3]. In general, pooling layers have two areas of application. First, by shrinking the spatial dimensions of the feature maps, pooling layers reduce the number of both the parameters and multiplications, improving the efficiency of deep neural networks. Second, pooling layers help to make their output feature maps approximately invariant towards small translations of the input feature maps.

Flattening Layers: Since convolutional layers compute a multidimensional output, it must be flattened to a one-dimensional vector for processing by a fully-connected layer. Thus, flattening layers are located between a convolutional and a fully-connected layer to reorganize the shape of the data.

2.1.2 Loss Functions

After each forward pass, a loss function $\mathcal{L}\left(\widehat{y}, y\right)$ calculates the difference between the model prediction $\widehat{y} = f_\Theta(x)$ and the known output y of the corresponding training example x. Depending on the type of learning task, various loss functions are employed. In the following, a brief description of the most common loss functions is given.

Mean squared error: The mean squared error (MSE) calculates the Euclidean distance between the prediction and target output and is therefore also known as the L^2-loss. It is

widely used for regression problems and is calculated as follows:

$$\mathrm{MSE}\left(\widehat{y}, y\right) = \|\widehat{y} - y\|^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\widehat{y}_i - y_i\right)^2 . \tag{2.13}$$

Here, N is the output dimension.

Mean absolute error: The mean absolute error (MAE) calculates the normalized sum of the absolute differences between the prediction and target output. It is also known as the L^1-loss and is mainly used for regression problems. The calculation is as follows:

$$\mathrm{MAE}\left(\widehat{y}, y\right) = \frac{1}{N}\sum_{i=1}^{N}\left|\widehat{y}_i - y_i\right| . \tag{2.14}$$

Categorical Cross Entropy: The cross entropy (CE) is a comparative measurement that indicates the similarity between two probability distributions. If the cross entropy is used in combination with a softmax output, one speaks about the categorical cross entropy loss. The latter is widely used for classification problems. The computation is as follows

$$\mathrm{CE}\left(\widehat{y}, y\right) = -\sum_{i=1}^{N} y_i \log\left(\widehat{y}_i\right) = -\sum_{i=1}^{N} y_i \log(\mathrm{softmax}(s)_i) . \tag{2.15}$$

Here, s is the output of the last layer before the softmax function has been applied. Since the categorical cross entropy loss measures the difference between two probability distributions, it is widely used for classification problems.

2.1.3 Backpropagation

Once the loss $\mathcal{L}\left(\widehat{y}, y\right)$ has been calculated, it is derived with respect to the network prediction $\widehat{y}$. According to Equation 2.2, the derivative can be written as

$$\frac{\partial \mathcal{L}\left(\widehat{y}, y\right)}{\partial \widehat{y}} = \frac{\partial \mathcal{L}\left(f^L\left(\ldots f^2\left(f^1\left(x_0, w_1\right), w_2\right), \ldots w_L\right), y\right)}{\partial \widehat{y}} . \tag{2.16}$$

Here, x_{l-1} is the input of layer f^l, and w_l indicates the set of parameters of layer f^l. Since each layer f^l forms a node of the computational graph that receives both w_l and x_{l-1} as input, the chain rule can be recursively applied to compute the gradient with respect to every variable of the graph. Consequently, the gradient for the set of learnable parameters w_l of layer l can be calculated as follows:

$$\frac{\partial \mathcal{L}\left(\widehat{y}, y\right)}{\partial w_l} = \frac{\partial \mathcal{L}}{\partial f^L\left(x_{L-1}, w_L\right)} \frac{\partial f^L\left(x_{L-1}, w_L\right)}{\partial f^{L-1}\left(x_{L-2}, w_{L-1}\right)} \cdots \frac{\partial f^{l+1}\left(x_l, w_{l+1}\right)}{\partial f^l\left(x_{l-1}, w_l\right)} \frac{\partial f^l\left(x_{l-1}, w_l\right)}{\partial w_l} \tag{2.17}$$

$$= \frac{\partial \mathcal{L}}{\partial\, \widehat{y}} \frac{\partial\, \widehat{y}}{\partial x_{L-1}} \frac{\partial x_{L-1}}{\partial x_{L-2}} \cdots \frac{\partial x_{l+1}}{\partial x_l} \frac{\partial x_l}{\partial w_l} . \tag{2.18}$$

Operation	Energy (pJ)	Relative (%)	Area Cost (μm^2)	Relative (%)
Add				
08-bit fixed-point	0.03	3	36	0.8
16-bit fixed-point	0.05	5	67	1.6
32-bit fixed-point	0.1	11	137	3.2
16-bit floating-point	0.4	44	1360	32
32-bit floating-point	0.9	100	4184	100
Mult				
08-bit fixed-point	0.2	5	282	4
16-bit fixed-point	-	-	-	-
32-bit fixed-point	3.1	83	3495	45
16-bit floating-point	1.1	30	1640	21
32-bit floating-point	3.7	100	7700	100

Table 2.1: A comparison between fixed-point and floating-point operations regarding their energy and area costs. All values correspond to an 45nm system with 0.9 V and are taken from [12, 30].

This makes it possible to reuse both the gradients already computed and the intermediate results of the forward pass to efficiently backpropagate the gradients even through complex models. In this way, it can be determined how the learnable parameters must be adjusted to reduce the learning loss. Consequently, the backpropagation algorithm is an application of the chain rule to efficiently compute the gradients of a computational graph with respect to its variables.

2.2 Fixed-Point Representation

Fixed-point numbers use a fixed partition of their bit size B for the integer part and the fractional part. In principle, the representation of a fixed-point number consists of an integer (sometimes also referred to as the mantissa) and a global scaling factor which can either be a power-of-two or a power-of-ten. In computer science, binary fixed-point numbers with a power-of-two scaling factor are widely preferred because rescaling operations can be efficiently implemented by shifting the decimal point accordingly.

On the one hand, the representation of an **unsigned fixed-point number** can be written as

$$x = m \times 2^{-f},\ m \in \mathbb{Z}_0^+,\ f \in \mathbb{Z}\,, \tag{2.19}$$

where m is the non-negative B-bit mantissa, 2^{-f} the global scaling factor, and f the position of the decimal point. If $B = 8$ bits and $f = 0$, the dynamic range is from 0 to 255. On the

other hand, there are several possibilities to represent signed fixed-point numbers that differ in their representation of the negative values. In the **sign-and-magnitude representation**, signed fixed-point numbers are described as follows:

$$x = (-1)^s \times m \times 2^{-f},\ s \in \{0, 1\},\ m \in \mathbb{Z}_0^+,\ f \in \mathbb{Z}\,. \tag{2.20}$$

Here, s is the sign bit, m is the non-negative $(B-1)$-bit mantissa, and f is the position of the decimal point. If $B = 8$ bits and $f = 0$, the dynamic range of the sign-and-magnitude representation ranges from –127 to 127, which creates an ambiguity of the zero value [31]. In contrast, if the **twos' complement** is used to represent negative values, the **representation** can be written as:

$$x = m \times 2^{-f},\ m \in \mathbb{Z},\ f \in \mathbb{Z}\,. \tag{2.21}$$

Here, m indicates a signed B-bit mantissa which is based on the twos' complement. If $B = 8$ bits and $f = 0$, the dynamic range of the twos' complement representation ranges from –128 to 127. Since the twos' complement offers unambiguousness for all numbers, it is commonly preferred in computer systems [31].

Compared to floating-point numbers, each of which has its separate scaling factor, fixed-point numbers share a global scaling factor over a given group of numbers. Considering the numerical representation of deep neural networks, a global scaling factor shared by all weights of a single layer can provide significant computational benefits. On the one hand, this allows the calculations of the respective layers to be parallelized very efficiently. On the other hand, if the global scaling factor is a power-of-two, multiplications with powers-of-two result in efficient bit shifts. In Table 2.2, a comparison between fixed-point and floating-point operations regarding their energy and area costs on a 45 nanometer (45 nm) lithography process is given. The values are taken from [30] and can also be found in [12]. In the case of additions, both the energy and area costs can be significantly reduced by using fixed-point instead of floating-point operations. For example, a 32-bit fixed-point add requires only 0.11 times as much energy as a 32-bit floating-point add does. In the case of multiplications, the main benefit is associated with a reduction of the bit size. For example, an 8-bit fixed-point multiplication requires only 0.064 times as much energy as a 32-bit fixed-point multiplication does. However, if the bit sizes of the operands and operations are to be reduced to less than 16-bits, both must be represented using fixed-point arithmetic.

In the underlying thesis, the proposed quantization methods use binary fixed-point numbers with power-of-two scaling factors. Furthermore, unsigned fixed-point numbers are represented as shown in Equation 2.19, whereas signed fixed-point numbers are represented using the twos' complement from Equation 2.21.

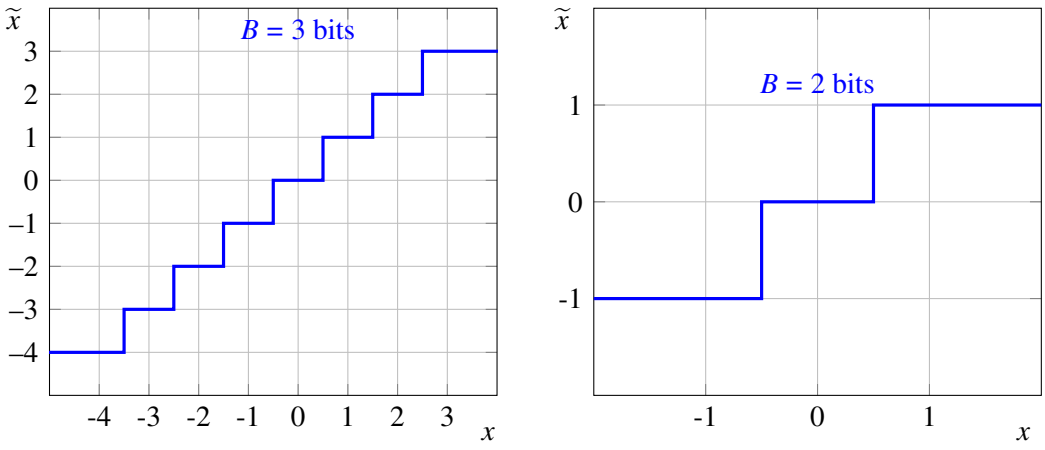

Figure 2.2: Illustration of two fixed-point quantization functions with the position of the decimal point at $f = 0$. On the left-hand side, a quantization function with $B = 3$ bits and a dynamic range from -4 to 3 is shown. On the right-hand side, a quantization function with $B = 2$ bits is shown that quantizes the input x to the ternary values, i.e. $\widetilde{x} \in \{-1, 0, 1\}$. In deep neural networks, ternary-valued weights offer the possibility of replacing many multiplications with additions. However, one possible quantization bin is lost.

2.3 Fixed-Point Quantization

A quantization function $Q : \mathbb{R} \rightarrow \{q_1, ..., q_K\}$ maps a real-valued input signal $x \in \mathbb{R}$ to a discrete output. Here, $\{q_1, ..., q_K\}$ is the set of discrete output values of size K, and B is the minimum number of bits necessary to represent all possible output values. Furthermore, if the quantization function is symmetric and has a uniform step size, the quantization can be calculated using scaling, rounding, and clipping functions. In this case, the zero-point remains unchanged, which is beneficial on computer systems [12, 18]. The computation of a symmetric and uniform quantization function can therefore be written as

$$\widetilde{x} = \begin{cases} Q^U(x, \Delta, B) = \text{clip}\left(\left\lfloor \frac{x}{\Delta} \right\rceil, 0, 2^B - 1\right)\Delta & \text{if } \widetilde{x} \text{ is unsigned}, \\ Q^S(x, \Delta, B) = \text{clip}\left(\left\lfloor \frac{x}{\Delta} \right\rceil, -2^{B-1}, 2^{B-1} - 1\right)\Delta & \text{if } \widetilde{x} \text{ is signed}. \end{cases} \tag{2.22}$$

Here, Q^U describes the unsigned quantization function, Q^S the signed quantization function, $\lfloor \cdot \rceil$ rounds to the closest integer, clip(z, min, max) truncates all values z to the domain [min, max], Δ is the step size, and B is the bit size of the quantized value $\widetilde{x}$.

Comparing the quantization functions from Equation 2.22 with the unsigned fixed-point representation from Equation 2.19 and the signed representation from Equation 2.21, it can be seen that the quantized value $\widetilde{x}$ is in fixed-point representation if and only of the step-size is a power-of-two. Hence, $\Delta = 2^{-f}$, $f \in \mathbb{Z}$ is the resulting fixed-point constraint with f

being the position decimal point of the quantized value $\widetilde{x}$. The corresponding fixed-point quantization functions can therefore be written as

$$\widetilde{x} = \begin{cases} Q^U(x, f, B) = \text{clip}\left(\left\lfloor \frac{x}{2^{-f}} \right\rceil, 0, 2^B - 1\right) 2^{-f} & \text{if } \widetilde{x} \text{ is unsigned,} \\ Q^S(x, f, B) = \text{clip}\left(\left\lfloor \frac{x}{2^{-f}} \right\rceil, -2^{B-1}, 2^{B-1} - 1\right) 2^{-f} & \text{if } \widetilde{x} \text{ is signed.} \end{cases} \tag{2.23}$$

2.3.1 Ternary-valued weights

However, there is a special case for ternary-valued weights. To replace many multiplications with additions, it is especially desirable to encode the layer weights with values from $\{-1, 0, 1\}$. Therefore, the 2-bit signed quantization function from Equation 2.23 can be modified to clip all values to the domain [-1, 1], which results in the set of quantized values $\{-2^{-f}, 0, 2^{-f}\}$. By factoring out the step-size of 2^{-f}, the layer weights can be encoded using binary values and the weighted sums that occur during the forward pass can be calculated using additions and subtractions. Afterwards, the step size is reintegrated by shifting the decimal point accordingly. Thus, ternary-valued weights are encoded using the following ternary quantization function Q^T:

$$\widetilde{x} = Q^T(x, f) = \text{clip}\left(\left\lfloor \frac{x}{2^{-f}} \right\rceil, -1, 1\right) 2^{-f}. \tag{2.24}$$

In Figure 2.2, a comparison between the 3-bit signed quantization function and the 2-bit ternary-valued quantization function is given. Due to the symmetric property of the clipping function, one quantization bin gets lost when quantizing the weights using ternary values.

2.3.2 Example: The Bit-shift Mechanism

This section provides an example to illustrate the computational path of a fixed-point quantization including the bit shift mechanism. The example is shown using a small fully-connected layer (see Equation 2.4) which is followed by a ReLU activation function (see Equation 2.11). For simplicity, the bias is assumed to be zero. Thus, the computation of the quantized layer can be written as:

$$\widetilde{x}_l = Q^U\left(\text{ReLU}\left(\widetilde{w}_l \widetilde{x}_{l-1}\right), f_{x,l} = 2, B = 4 \text{ bits}\right) \quad \text{with} \quad \widetilde{w}_l = Q^T\left(w_l, f_{w,l} = 1\right). \tag{2.25}$$

Here, $\widetilde{x}_l$ is the input of layer l which is quantized to $B = 4$ bits using the unsigned fixed-point quantization function Q^U from Equation. 2.23, and $\widetilde{w}_l$ is the ternary-valued weight matrix of layer l which is quantized using $B = 2$ bits and the ternary quantization function Q^T from Equation 2.24. Furthermore, $f_{x,l} = 2$ is the position of the decimal point of the quantized activations $\widetilde{x}_l$, and $f_{w,l} = 1$ is the position of the decimal point of the quantized weight matrix $\widetilde{w}_l$.

In Figure 2.3, the computational path is shown for exemplary weights and input values: The upper part shows intermediate results in the decimal notation whereas the lower part provides the corresponding binary code. According to Section 2.3.1, ternary-valued weights can be decomposed into binary values $\{-1, 0, 1\}$ and the layer-specific step size $2^{-f_{w,l}}$, with $f_{w,l}$ being the position of the decimal point of the weights in layer l. In the underlying example, the fully-connected layer consists of three neurons, resulting in a 3×3 weight matrix. Since the decomposed weight values are either 0 or ± 1, multiply-and-accumulate operations that occur during the forward pass $\left(\text{by calculating } \widetilde{w}_l \widetilde{x}_{l-1}\right)$ can be computed using additions and subtractions, depending on whether the corresponding weight value is negative or positive. Thus, the corresponding computation block is marked with ADD in Figure 2.3. After accumulating the input values, the step size $2^{-f_{w,l}}$ is reintegrated by shifting the decimal points $f_{w,l} = 1$ positions to the left (see the Shift block at the top in Figure 2.3).

Next, the ReLU activation function truncates all negative input values (see $\geq$ in Figure 2.3). Subsequently, the unsigned fixed-point quantization function Q^U quantizes the layer activation using $B = 4$ bits. Here, the decimal point of the quantized activations is $f_{x,l} = 2$. Therefore, the decimal point is shifted two positions to the right (see the Shift block at the bottom left in Figure 2.3), which is equivalent to dividing by 2^{-2}. The following rounding can be implemented as follows: If there is an active bit directly to the right of the decimal point, the corresponding value is rounded up (which is the case for the upper two values). If this is not the case, the value is rounded down. The following clipping function takes into account the quantization domain consisting of the $B = 4$ bits on the left side of the decimal point (which are marked in blue in Figure 2.3). If there is another active bit to the left of the quantization domain, the value is clipped by activating all four bits of the quantization domain. In the underlying example, this is the case for the topmost value. Since all negative values have already been clipped by the ReLU activation function, the decimal point is now shifted back two positions to the right and the quantization is finished.

Notice: This is only a semantic representation to demonstrate the bit shift mechanism using fixed-point arithmetic. Both the dimensions and the values are arbitrarily selected. Furthermore, the ReLU and the unsigned fixed-point quantization can be converted into a single function (since the unsigned quantization function truncates all negative activations). However, to make the example more comprehensible, both functions were calculated separately here.

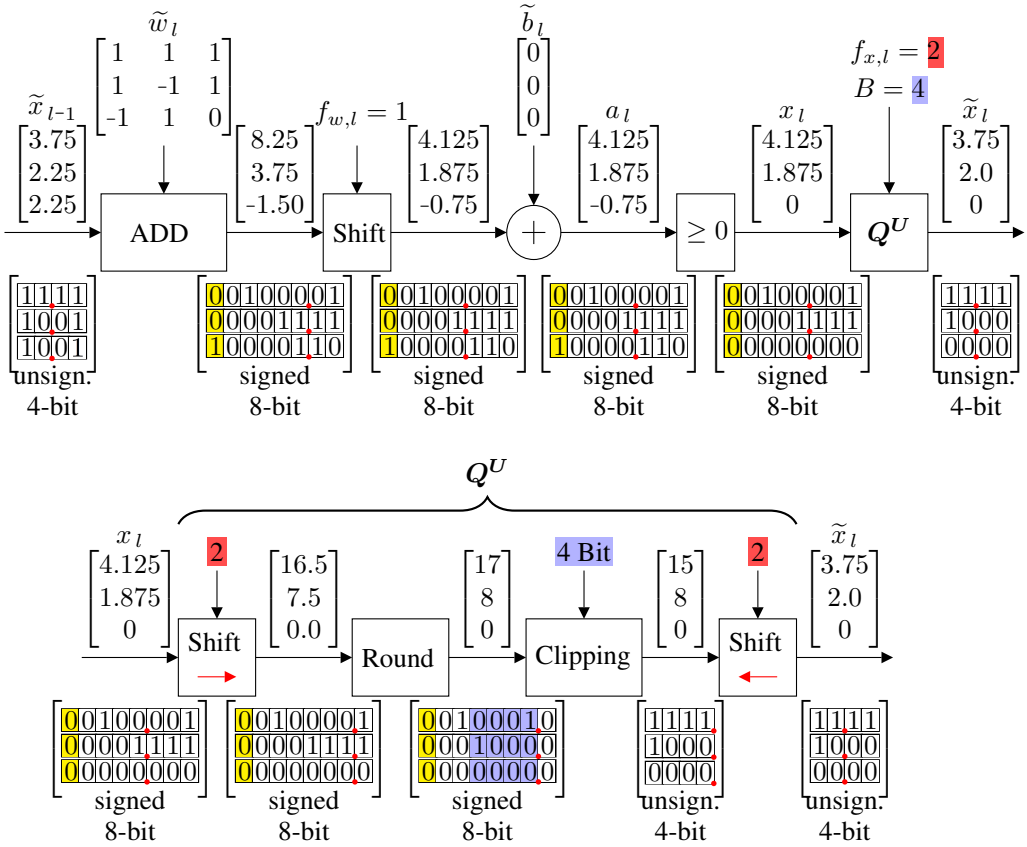

Figure 2.3: Computational path of a small fully-connected layer that can be evaluated using pure fixed-point arithmetic. The activations $\widetilde{x}_{l-1}$ and $\widetilde{x}_l$ are quantized using the unsigned fixed-point quantization function Q^U from Equation 2.23 with $B = 4$ bits. The layer weights $\widetilde{w}_l$ are quantized using the ternary quantization function Q^T from Equation 2.24. The example shows how the forward pass can be computed without the need for multiplications. First, the ternary-valued weights can be decomposed into binary values and the layer-specific step size which is a power of two whose exponent indicates the position of the decimal point. As a result, multiplying the binary weights with the quantized input can be done using additions and subtractions, depending on whether the corresponding weight value is negative or positive. Subsequently, the decimal point is shifted according to the exponent of the respective step size (the so-called bit shift mechanism). Next, the ReLU activation function truncates all negative input values. Subsequently, the unsigned fixed-point quantization function quantizes the activations to $B = 4$ bits by the following steps: 1.) Dividing by the step size $2^{-f_{x,l}}$ results in shifting the decimal point $f_{x,l} = 2$ positions to the right. 2.) The values are rounded by considering the bit directly to the right of the decimal point. 3.) All values are clipped according to the quantization domain, which consists of $B = 4$ bits and is marked in blue. If there is an active bit to the left of the quantization domain, the respective value is clipped by activating all bits of the quantization domain (which is the case for the topmost value). 4.) Subsequently, the decimal point is shifted backward two positions to the left.

Chapter 3

Data Sets and Architectures

This section introduces both the data sets and the network architectures that are used for the experimental evaluation of this thesis.

3.1 MNIST

MNIST is a handwritten-digits classification task with 28×28 gray scale images [32]. Here, each image contains a handwritten number between 0 and 9, which is to be classified correctly by a computer vision algorithm. The data set is divided into 60,000 training and 10,000 test samples. We preprocess the data by subtracting the mean and dividing it by the standard deviation over the training set.

Architectures: For the MNIST classification task, we use the **LeNet-5** architecture from [33]. The composition of the layers is as follows:

$$\text{C6-5x5} + \text{SS2} + \text{C16-5x5} + \text{SS2} + \text{FC120} + \text{FC84} + \text{FC10}\,. \tag{3.1}$$

Here, C6-5x5 represents a convolutional layer with six kernel filters of size 5x5 each, FC120 is a fully-connected layer with 120 output neurons, and SS2 is a sub-sampling module that reduces the spatial dimensions of the corresponding output feature maps by two. Each layer except the last uses a hyperbolic tangent activation function. In contrast, the output layer is followed by a softmax activation function.

3.2 CIFAR-10

CIFAR-10 is an image classification task with 10 different classes [34]. The data consists of 32 × 32 colored natural images and is divided into 50,000 training and 10,000 test samples. We preprocess the images as recommended in [35]: We normalize the data by subtracting the channel means and dividing it by the standard deviations. Furthermore, we use the standard data augmentation by applying random horizontal flips and shifting [35].

Architectures: For the CIFAR-10 classification task, we evaluate different network architectures. On the one hand, we use the **VGG7** architecture proposed in [36]. The layer composition is as follows:

$$2\text{ C128-3}\times3 + \text{MP2} + 2\text{ C256-3}\times3 + \text{MP2} + 2\text{ C512-3}\times3 + \text{MP2} + \text{FC1024} + \text{FC10}\,. \tag{3.2}$$

Here, 2 C128-3x3 represents two convolutional layers with 128 kernel filters of size 3x3 each, FC1024 is a fully-connected layer with 1024 output neurons, and MP2 is a max-pooling layer that reduces the spatial dimensions of the corresponding feature maps by two. Each layer except the last is followed by a batch-normalization layer and a ReLU activation function. The output layer is followed by a softmax activation function.

On the other hand, we use **ResNet-20** and **ResNet-56**. A description of both the implementation details and the layer compositions can be found in [5] as well as in the PyTorch model zoo [37]. Compared to LeNet-5 and VGG7, which are conventional feed-forward neural networks, ResNet architectures use shortcut connections between single layers that sum up the respective output feature maps. The residual functions learned this way improve the gradient flow during backpropagation and thus increase the performance of deep network architectures [5].

In addition, we use **DenseNet** with 76 layers and a growth rate of 12 (L = 76, k = 12, [35]). In DenseNet architectures, each layer receives the output feature maps of all preceding layers as input. This reduces the vanishing gradient problem, improves the reuse of features learned, and decreases the number of parameters [35]. Due to its lower number of redundancies, DenseNet is described as difficult to reduce [38]. The implementation details can be found in [35].

3.3 CIFAR-100

CIFAR-100 uses the same images as CIFAR-10 but provides 10 additional sub-classes for each class in CIFAR-10. Thus, only 500 training samples and 100 test samples are available for each class, which makes CIFAR-100 a challenging classification task. We use the same preprocessing steps and data augmentation as in the case of CIFAR-10.

Architectures: For the CIFAR-100 image classification task, we use **VGG11** and **VGG16**. Both network architectures are similar to VGG7 but use 11 and 16 layers, respectively. The computational graph of **VGG11** is as follows:

$$\begin{aligned}&\text{C64-3x3} + \text{MP2} + \text{C128-3x3} + \text{MP2} + 2\text{ C256-3x3} + \text{MP2} + 2\text{ C512-3x3} + \text{MP2} + \\ &2\text{ C512-3x3} + \text{MP2} + \text{FC4096} + \text{FC4096} + \text{FC100}\,.\end{aligned} \tag{3.3}$$

Here, the notation is the same as for VGG7. Each layer except the last is followed by a batch-normalization layer and a ReLU activation function. The output layer is followed by a softmax activation function. The computational graph of **VGG16** is as follows:

$$\text{2 C64-3x3 + MP2 + 2 C128-3x3 + MP2 + 3 C256-3x3 + MP2 + 3 C512-3x3 + MP2 + 3 C512-3x3 + MP2 + FC4096 + FC4096 + FC100}\,. \tag{3.4}$$

The notation is the same as for VGG7 and VGG11. Each layer except the last is followed by a batch-normalization layer and a ReLU activation function. The output layer is followed by a softmax activation function.

3.4 ImageNet

ImageNet is a data set consisting of real-world color images. It is used for image classification tasks and provides 1,000 class labels. We use ILSVRC12 from [39], which consists of 1,281,167 training and 50,000 test samples. The data is preprocessed by subtracting the mean and dividing it by the standard deviation over the training set. If data augmentation is applied, the data is mutated by applying random horizontal flips and random cropping to the size 224×224.

Architectures: For the ImageNet classification task, we use different network architectures including **ResNet-18** and **ResNet-50**. The implementation details and the compositions of layers can be found in [5] as well as in the PyTorch model zoo [37]. Furthermore, we use **MobileNet-V1** and **MobileNet-V2** [40]. MobileNet architectures apply depth-wise separable convolutions, which reduces both the computational and memory requirements [40]. The implementation details and the composition of layers can be found in [40] as well as in the PyTorch model zoo [37]. In addition, we use **InceptionV3** [41].

Chapter 4

Network Quantization

Deep neural networks have proven to outperform classical methods on several machine learning benchmarks. However, they have high computational complexity and require powerful processing units. Especially when deployed on embedded systems, both their memory requirements and inference time must be significantly reduced. Therefore, we propose SYMOG (symmetric mixture of Gaussian modes), a novel soft quantization approach that significantly decreases the complexity of deep neural networks through low-bit fixed-point quantization. SYMOG uses a reduction loss to train deep neural networks with minimal quantization error. After training, the distribution of the weights resembles a multi-model distribution, with each mode corresponding to a certain fixed-point number. We evaluate our approach by quantizing the weights using ternary values. Thus, most multiplications required during the forward pass can be replaced by additions. Compared to related quantization approaches, we achieve new state-of-the-art performance with the lowest number of training epochs.

Furthermore, we propose EEquant (easy and effective quantization), an approach to train deep neural networks that can be evaluated using pure fixed-point arithmetic afterwards. EEquant is an extension of our SYMOG soft quantization approach that takes into account the distribution of the parameters of the batch-normalization layers. Furthermore, by using a discrete ReLU activation function, EEquant quantizes the layer activations during each forward pass. After training, the batch-normalization layers are folded into the preceding convolutional or fully-connected layers and the model can be evaluated without the need for floating-point multiplications.

4.1 Introduction

Deep neural networks are state-of-the-art in many machine learning problems, enabling recent progress in computer vision, speech recognition, and object detection [2, 42, 43]. However, the best results have been accomplished by training large models with many parameters using large amounts of training data [2, 43]. As a result, modern deep neural networks have an extensive memory footprint, with floating-point multiplications being especially expensive in terms of computation time and power consumption [12]. When deployed on embedded devices, deep neural networks are necessarily restricted by their computational complexity.

Therefore, various approaches have been developed to optimize deep neural networks to better match embedded hardware constraints [13, 18, 44, 45]. One of these approaches involves reducing the bit sizes of the parameters and activations through quantization [15, 16, 36, 46, 47, 48]. On ordinary computational hardware, the quantization of the parameters reduces the memory requirements, as well as the amount of data accesses [12]. Furthermore, ternary- or even binary-valued parameters replace many multiplication with additions [36, 46, 49]. On dedicated fixed-point hardware, the quantization of both the parameters and activations using fixed-point representations simultaneously reduces memory cost, inference time, and energy consumption [13, 18].

Since post-training quantization usually fails to maintain the accuracy of deep neural networks with floating-point precision [48], quantization-aware training methods have been developed to integrate fixed-point constraints into the training of deep neural networks [13, 18]. These quantization-aware training methods are mainly grouped into two categories. On the one hand, hard quantization methods use quantized parameters (and optionally even quantized activations) during both the forward and backward pass. Thus, the quantization noise is taken into account during training and the parameters can be adjusted accordingly. However, since the derivative of the rounding function is zero almost everywhere, the local gradients of the applied quantization functions must be estimated during the backward pass, which creates gradient mismatches. On the other hand, soft quantization approaches use floating-point parameters during the training but simultaneously promote posterior distributions of the parameters that are well qualified for post quantization. This is done by including additional constraints (e.g. additional loss functions or regularization terms) into the training of deep neural networks such that both the learning loss and the quantization noise are minimized simultaneously. Thus, soft quantization allows training in floating-point precision without the need for gradient estimators. After the training, the parameters can ideally be quantized with no significant loss in accuracy.

However, previous approaches have several drawbacks. The quantization functions used in [20, 36, 46, 47, 49] apply high-precision scaling factors, which exclude the possibility of pure fixed-point arithmetic on dedicated hardware. In [36, 46, 47, 49], the batch-normalization layers are not integrated into the fixed-point representation of

the parameters of the preceding layers and thus remain in floating-point precision. In [13, 18, 50], the computational graph must be changed significantly to calculate first the batch statistics of the layer activations and subsequently the quantized output of the folded batch normalization layers. This increases the computational effort during training, and the quantized weights start to jitter [50]. As a result, several modifications must be made to stabilize the training procedure and increase the test performance, such as freezing batch-normalization layers and the correction of quantized weights [18, 50]. All these methods and their modifications are discussed in detail in Section 4.4.

In this chapter, we propose the following quantization approaches:

- In Section 4.2, we propose SYMOG, a novel soft quantization approach to train deep neural networks with minimal quantization error. With SYMOG training, the distribution of the weights changes from and uni-modal to a symmetric and multi-modal distribution as shown in Figure 4.1. Here, the center of each mode corresponds to a certain fixed-point number. After training, the weights can be quantized into a fixed-point representation with no significant loss in accuracy. Since only the reduction loss representing the expected quantization error has to be integrated, SYMOG is easy to implement and does not require gradient estimators.

  We evaluate our SYMOG approach with the special case of ternary-valued weights. Due to the fixed-point constraint, the ternary weights can be decomposed into binary values and a power-of-two scaling factor. Thus, most of the multiplications that are required during the forward pass can be replaced by additions. Furthermore, scaling with a power-of-two results in a simple bit shift.

- In Section 4.3, we propose EEquant by extending our SYMOG soft quantization approach towards batch-normalization. After training with EEquant, the batch-normalization layers can be folded into the preceding convolutional or fully-connected layers, with the parameters of the folded layers resembling a symmetric and multi-modal distribution. Consequently, the folded layers can be converted into a fixed-point representation with no significant loss in accuracy. Furthermore, by quantizing the layer activations using a discrete ReLU activation function, the trained networks can be evaluated using pure fixed-point arithmetic.

4.2 SYMOG: Symmetric Mixture of Gaussian Modes

The training of deep neural networks works best using floating-point precision. However, in soft quantization, the training must promote posterior distributions of the parameters that can be quantized with no significant loss in accuracy afterwards. Therefore, we first introduce a reduction loss whose minimization results in a distribution of the weights well suited for post-quantization using fixed-point arithmetic. Furthermore, we give

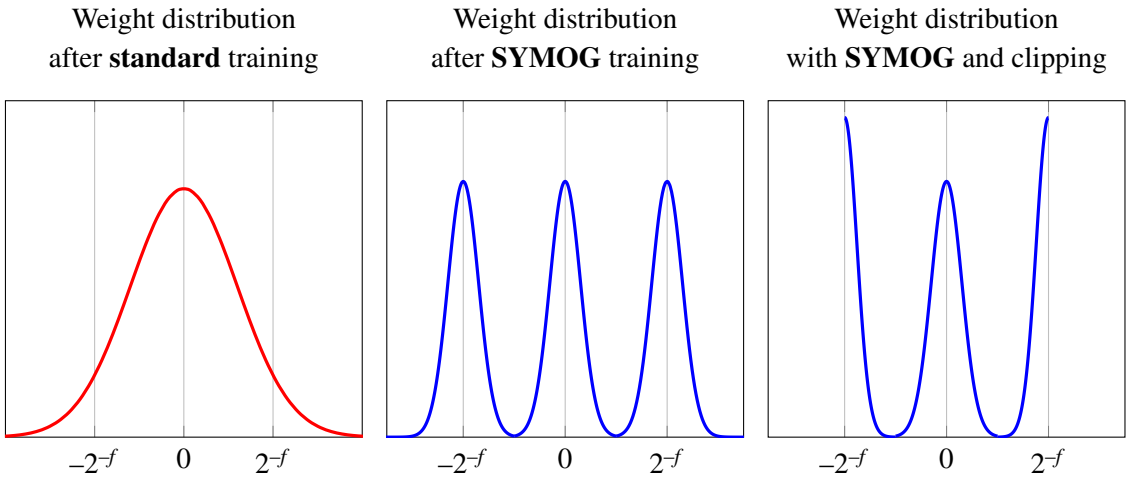

Figure 4.1: A simplified comparison of different weight distributions after training. After conventional training, the weights are usually uni-modal Gaussian distributed. Thus, a quantization to symmetric bins results in a high quantization error and poor performance (left). Here, 2^{-f} is the uniform step-size of the quantization function, f is the position of the decimal point of the fixed-point representation, and $\{-2^{-f}, 0, 2^{f}\}$ is the corresponding set of quantization bins. In contrast, with SYMOG, each quantization bin is represented by a single Gaussian distribution such that both the quantization error (i.e. the variance of the Gaussian modes) as well as the learning loss can be minimized simultaneously during training. The resulting weight distribution is multi-modal Gaussian distributed (middle) and yields only a small quantization error. Furthermore, the weight adaptation can be improved by clipping the weights according to the quantization domain (right).

implementation details that improve the weight adaptation during training. We evaluate our approach with symmetric 2-bit weights, which significantly improves both the memory requirements and the computational complexity during inference.

4.2.1 Reduction Loss

According to Section 2.1, a deep neural network is a computational graph f_Θ parameterized by a set of parameters $w \in \Theta$, $\Theta = \{w_1, ..., w_M\}$ of size M. The network is trained by minimizing a learning loss $\mathcal{L}_{\text{learn}}\left(\widehat{y}, y\right)$ with respect to its set of parameters Θ. Here, $\mathcal{L}_{\text{learn}}$ calculates the difference between the network prediction $\widehat{y} = f_\Theta(x)$ and the true output y of a corresponding training example x. To influence the distribution of the parameters, a regularization term $R : \mathbb{R}^M \rightarrow \mathbb{R}$ can be added to the learning loss that takes as input the set parameters Θ. Thus, the training represents a trade-off between fulfilling the learning task and regularizing the distribution of the weights.

A widely used regularization for preventing high weight values is the L^2-norm [51],

which is applied to the entire set of trainable parameters as follows:

$$R_{\mathrm{L}^2} = \sum_{m=1}^{M} \|w_m\|_2^2 = \sum_{m=1}^{M} w_m^2 \,. \tag{4.1}$$

Training with the L^2-regularization penalizes the squared absolute values of the weights and usually results in a uni-modal Gaussian distribution of the parameters, as can be seen in Figure 4.1 on the left. In order to combine multiple Gaussian modes and the fixed-point representation stated in Section 2.2, we propose the following L^2-norm based reduction loss:

$$\mathcal{L}_{\text{reduce}} = \sum_{l=1}^{L} \sum_{i=1}^{M_l} \frac{1}{M_l} \left\| w_{l,i} - Q^S\left(w_{l,i}, f_l, B\right) \right\|_2^2 = \sum_{l=1}^{L} \sum_{i=1}^{M_l} \frac{1}{M_l} \left(w_{l,i} - \widetilde{w}_{l,i}\right)^2 \,. \tag{4.2}$$

Here, L is the number of layers, M_l the number of weights in layer l, $\widetilde{w}_{l,i}$ the quantized weight value, $f_l \in \mathbb{Z}$ the position of the decimal point of the fixed-point representation in layer l, B the bit size, and Q^S the signed fixed-point quantization function from Equation 2.23.

Effectively, the reduction loss gives individual Gaussian priors to each network weight by calculating the mean squared quantization error for each network layer. The priors are updated during each forward pass with respect to the closest fixed-point mode, enabling the weights to continuously switch between neighboring modes. The gradient of the reduction loss with respect to a certain weight value $w_{l,i}$ is as follows:

$$\frac{\partial \mathcal{L}_{\text{reduce}}}{\partial w_{l,i}} = \frac{2}{M_l} \left(w_{l,i} - \widetilde{w}_{l,i}\right) \left(1 - \frac{\partial \widetilde{w}_{l,i}}{\partial w_{l,i}}\right) = \begin{cases} \pm\infty \text{ if } w_{l,i} = \left(k + \frac{1}{2}\right) 2^{-f_l}, k \in \mathbb{Z} \\ \frac{2}{M_l} \left(w_{l,i} - \widetilde{w}_{l,i}\right) \text{ else} \end{cases} \,. \tag{4.3}$$

The derivative of the quantization function is zero except at the threshold values of the rounding function, see Figure 2.2. However, due to real-valued layer weights, this case can be neglected, and the partial derivative of the quantization function can be calculated with zero for all input values. Thus, the gradient is a scaled version of the corresponding quantization error, see case two in Equation 4.3. Updating the weights in the direction of their negative gradients decreases the quantization error and approximates the fixed-point representation of the weights.

During training, the reduction loss is added to the learning loss $\mathcal{L}_{\text{learn}}$ according to

$$\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda \, \mathcal{L}_{\text{reduce}} \,. \tag{4.4}$$

Here, $\mathcal{L}_{\text{train}}$ is the overall training objective, and λ is the regularization parameter that controls the weighting between both losses: the larger λ, the higher the contribution of the reduction loss whose minimization results in a reduction of the model capacity. To increase the model capacity at the beginning of the training, we start with a small value of λ and increase it as the training progresses. Therefore, we recommend exponential growth as described in Section 4.2.3.

4.2.2 Weight Clipping

Soft quantization approaches use regularization terms or additional loss functions during the training of deep neural networks to promote posterior distributions of the parameters that are well qualified for post quantization. Since SYMOG is a fixed-point quantization approach, the potential solution space of the layer weights is limited to the quantization domain $\left[2^{-f_l}\ \left(-2^{B-1}\right), 2^{-f_l}\ \left(2^{B-1}-1\right)\right]$, see Equation 2.22. Consequently, weights should not exceed this interval during training. For example, a configuration with $B = 3$ bits and $f_l = 0$ leads to the following set of quantized values: $\{-4, -3, .., 2, 3\}$. Once a weight takes the value –4, it is useless to update it in the negative direction because it would diverge from the solution space. The same concept applies on the opposite side. Therefore, we clip all weights to the quantization domain $\left[2^{-f_l}\ \left(-2^{B-1}\right), 2^{-f_l}\ \left(2^{B-1}-1\right)\right]$ after each update step to promote constructive weight adaptations. Except in the case of quantizing the weights using ternary-values, when we clip to the quantization domain $\left[-2^{-f_l}, 2^{-f_l}\right]$. A simplified visualization of the clipping procedure is given in Figure 4.1 on the right-hand side. We will also demonstrate the benefit of the weight clipping in the experimental section.

4.2.3 Implementation Details

Algorithm 1 summarizes our SYMOG approach to train a deep neural network with multi-modal weight distributions that enable an accurate fixed-point quantization after the training. The pretrained model f_Θ, the number of training epochs E, the training set $D = \{(x_i, y_i)\}_{i=1}^{d}$, the batch size S, the learning loss $\mathcal{L}_{\text{learn}}$, the learning rate domain $[\eta_1, \eta_E]$, the start value λ_0 as well as the growth-factor α_E of the regularization parameter, and the bit size B are required as input values. Regarding the learning rate and the regularization parameter, we recommend using $[\eta_0, \eta_E] = [0.01, 0.001]$, $\lambda_0 = 10$, and $\alpha_E = 9/E$. Furthermore, if $B = 2$ bits, the weights are quantized using the ternary quantization function Q^T from Equation 2.24 (see line 3). Otherwise, the signed fixed-point quantization function Q^S from Equation 2.23 is used (see line 5).

First of all, the layer-wise decimal points are initialized such the quantization error of the pretrained weights is minimized (see lines 7 to 10). Since the decimal points are integer values, the corresponding optimization problem is discrete and can be solved by a grid search. Once the decimal points have been initialized, SYMOG training starts. At the beginning of each epoch, both the learning rate and the regularization parameter are scheduled. Here, we linearly decrease the learning rate from η_0 to η_E (see line 12), whereas the regularization parameter is exponentially increased over the training epochs (see line 13). Furthermore, the training data is shuffled and divided into N mini-batches of size S. Here, we use a batch size of 128.

A single training step is implemented as follows. First, the respective mini-batch is propagated through the model to compute the model prediction (see line 17). Next, both

Algorithm 1 SYMOG: Training a deep neural network with minimal quantization error. During training, the distribution of the weights changes from a uni-modal distribution to a multi-modal distribution with each mode corresponding to a fixed-point number. After training, the weights can be quantized with no significant loss in accuracy.

Input: Pretrained model f_Θ, Number of Epochs E, Training Data $D = \{(x_i, y_i)\}_{i=1}^{d}$, Batch size S, Learning loss $\mathcal{L}$, Learning-rate domain $[\eta_0, \eta_E]$, Regularization start value λ_0 and growth-factor α_E, Desired bit size B.
if $B = 2$ **then**
    $Q \leftarrow Q^T$
else if $B > 2$ **then**
    $Q \leftarrow Q^S$
end if
for $l = 1$ to L **do**
    min. $\|w_l - Q(w_l, f_l, B)\|^2$
    s.t. $f_l \in Z$
end for
for $e = 1$ to E **do**
    $\eta \leftarrow \eta_0 - (\eta_0 - \eta_E)\, e/E$
    $\lambda \leftarrow \lambda_0 \cdot \exp(\alpha_E\, e)$
    Randomly shuffle D.
    Divide D into N batches $\{X_n, Y_n\}_{n=1}^{N}$ of size S.
    for $n = 1$ to N **do**
        $\widehat{Y}_n \leftarrow f_\Theta(X_n)$
        Compute $\mathcal{L}_{\text{learn}}\left(\widehat{Y}_n, Y_n\right)$
        Compute $\mathcal{L}_{\text{reduce}}$ according to Equation 4.2
        Compute $\mathcal{L}_{\text{train}} \leftarrow \mathcal{L}_{\text{learn}} + \lambda\, \mathcal{L}_{\text{reduce}}$
        Compute $\frac{\partial \mathcal{L}_{\text{train}}}{\partial w}$
        $w \leftarrow w - \text{SGD}\left(w, \eta, \frac{\partial \mathcal{L}_{\text{train}}}{\partial w}\right)$
        for $l = 1$ to L **do**
            $w_l \leftarrow \text{clip}\Big(w_l,\, 2^{-f_l}\left(-2^{B-1}\right),\, 2^{-f_l}\left(2^{B-1} - 1\right)\Big)$
        end for
    end for
end for
for $l = 1$ to L **do**
    $\widetilde{w}_l \leftarrow Q(w_l, f_l, B)$
end for
return $\{\widetilde{w}_l\}_{l=1}^{L}$

the learning loss and the reduction loss are calculated (see lines 18 and 19) and added together to form the training loss (see line 20). Subsequently, the training loss is derived with respect to the model parameters (see line 21), and each weight is updated in the direction of its negative gradient. Here, we use stochastic gradient descent optimization (SGD [28]) with the Nesterov momentum set to 0.9 (see line 22). After the update step, the layer weights are clipped according to their quantization domain (see line 24). This is repeated until the number of epochs E is reached. After the training, all weights are quantized with respect to the closest fixed-point mode (see line 29).

4.2.4 Experiments and Results

We evaluate our SYMOG approach on the benchmark data sets MNIST, CIFAR-10, and CIFAR-100. A detailed description of the datasets and architectures can be found in Section 3. We quantize the weights to power-of-two ternary values $\widetilde{w} \in \{-2^{-f}, 0, 2^{-f}\}$ as shown in Figure 4.1 by using the fixed-point quantization function Q^T from Equation 2.24. Thus, the network weights can be encoded using binary values $\{-1, 0, 1\}$ and a power-of-two scaling factor. As a result, most of the multiply-and-accumulate operations that are required during the forward pass can be replaced by additions and subtractions [36, 46, 47]. Furthermore, scaling by a power-of-two can be computed by shifting the decimal point accordingly. In this way, ternary-valued weights combined with power-of-two scaling factors can significantly reduce the memory requirements and the computational effort during inference.

We organize our experiments in two parts. First, we show state-of-the-art performance on different classification tasks by quantizing different network architectures, from small- to large-scale. The results are shown in Table 4.1. Here, we compare our SYMOG approach with weight compression results from BinnaryConnect (BC, [46]), Ternary Weight Networks (TWN, [36]), Variational Network Quantization (VNQ, [38]), and BinaryRelax (BR, [52]). We also show that our method requires significantly fewer training epochs than competitive approaches. Second, we give insights into the training procedure and illustrate the adaptation of the weights during training. All experiments are done using Algorithm 1.

MNIST with LeNet-5: For the MNIST classification task, we use the LeNet-5 architecture and train for 25 epochs. Our approach achieves a Top-1 accuracy slightly higher than the baseline accuracy (99.37% vs. 99.30% baseline accuracy). This may be due to the reduction loss, which can behave like regularization and thus improve the generalization capability on simple classification tasks. Compared to TWN and VNQ, which both quantize the weights to ternary values, our approach slightly improves the Top-1 accuracy (99.37% vs. 99.35% and 99.27%, respectively) and significantly reduces the number of training epochs (25 vs. 40 and 195, respectively). However, both TWN and VNQ use real-valued scaling factors, making it impossible to encode the weights using fixed-point arithmetic.

Data set	Method	Model	#Parameter	Bits	FP	Epochs	Top-1 [%]
MNIST	*Baseline*	LeNet5	60k	32	○	25	99.30
	BC [46]	-	-	1	●	-	98.71
	TWN [48]	LeNet5	60k	2	○	40	99.35
	VNQ [38]	LeNet5	60k	2	○	195	99.27
	SYMOG	LeNet5	60k	2	●	25	**99.37**
CIFAR-10	*Baseline*	VGG7	12M	32	○	100	94.48
	BC [46]	VGG8	14M	1	●	500	90.10
	TWN [48]	VGG7	12M	2	○	170	92.56
	SYMOG	VGG7	12M	2	●	100	**94.29**
	Baseline	DenseNet	0.49M	32	○	100	94.28
	VNQ [38]	DenseNet	0.49M	2	○	150	91.17
	SYMOG	DenseNet	0.49M	2	●	100	**94.04**
CIFAR-100	*Baseline*	VGG11	32M	32	○	100	68.58
	TWN [48]	VGG11	32M	2	○	300	63.82
	BR [52]	VGG11	32M	2	○	300	65.87
	SYMOG	VGG11	32M	2	●	100	**67.95**
	Baseline	VGG11	34M	32	○	100	73.42
	TWN [48]	VGG16	34M	2	○	300	71.41
	BR [52]	VGG16	34M	2	○	300	72.10
	SYMOG	VGG16	34M	2	●	100	**72.35**

Table 4.1: Results of different quantization methods on MNIST, CIFAR-10, and CIFAR-100. The table indicates the model name, the number of parameters of the respective model, the number of bits used for quantization, the number of training epochs, as well as the test accuracy. Furthermore, column six indicates whether the respective method fulfills the fixed-point (FP) constraint of power-of-two scaling factors: ○ = False, ● = True. The baseline gives the 32-bit floating-point accuracy. Our SYMOG approach yields both the highest Top-1 accuracies and the smallest number of training epochs.

On the other hand, BC uses binary values to quantize the weights without using scaling factors, which can be highly efficient on dedicated hardware. However, the Top-1 accuracy drops below 98%, and the model used in BC consists of large fully-connected layers with significantly more parameters. Thus, it is much easier to quantize compared to LeNet-5.

CIFAR-10 with VGG7 & DenseNet: For the CIFAR-10 classification task, we evaluate two different network architectures. First, we test VGG7 by training for 100 epochs. With ternary-valued weights, our approach achieves a Top-1 accuracy that is only 0.2% below the baseline accuracy (94.29% vs. 94.48% baseline accuracy). Compared to TWN, our approach improves the Top-1 accuracy by 1.7% with a simultaneous reduction in the number of training epochs from 170 to 100. Compared to BC, our approach improves the Top-1 accuracy by 4% with only one-fifth of the training epochs. Second, we evaluate DenseNet, which has an optimized architecture with comparatively few parameters. Due to its lower number of redundancies, DenseNet is described as difficult to quantize [38]. However, after 100 epochs of training, our approach achieves a 2-bit performance that is only 0.24% below the baseline accuracy (94.04% vs. 94.28% baseline accuracy). Compared to the Bayesian approach of VNQ, our approach improves the Top-1 accuracy by nearly 3% with a simultaneous reduction of the number of training epochs from 150 to 100.

CIFAR-100 with VGG11 & VGG16: CIFAR-100 uses the same images as CIFAR-10 but provides 10 additional sub-classes for each class in CIFAR-10. Thus, only 500 training samples and 100 test samples are available for each class, which makes CIFAR-100 a challenging classification task. On the one hand, we evaluate with VGG11. After 100 epochs of training, our approach achieves a Top-1 accuracy which is only 0.63% below the baseline (67.95% vs. 68.58% baseline accuracy). Thus, we outperform BR by more than 2% (67.95% vs 65.87%) and TWN by more than 4% (67.95% vs. 63.82%). Furthermore, our approach significantly reduces the number of training epochs from 300 to 100. On the other hand, we evaluate with VGG16. Here, our approach performs best using symmetric ternary-valued weights as well. The Top-1 accuracy is slightly higher compared to BR (72.35% vs 72.10%) at one-third of the training time (100 vs 300 epochs).

In summary, our soft quantization approach outperforms the hard quantization approaches BC, TWN, and BR in both training time and accuracy. Especially, the number of training epochs is significantly reduced by a factor of up to five. This may be due to our soft quantization approach: In contrast to hard quantization, the training remains in floating-point precision. Consequently, there are no gradient mismatches due to gradient estimators and training converges better. Compared to VNQ, which is a Bayesian soft quantization approach, our approach increases performance substantially using the small DenseNet architecture.

Weight adaptation: SYMOG is a soft quantization approach to train deep neural networks with multi-modal weight distributions, where each mode corresponds to a certain fixed-point number. This is done by minimizing both the learning loss and the quantization error simultaneously during training. After training, the weights are quantized concerning their closest fixed-point number. In this section, we give insights into the training and visualize the adaptation of the weights using the example of VGG11 (the corresponding Top-1 accuracy is shown in Table 4.1).

On the one hand, Figure 4.2 shows the weight distributions of the layers with index 1, 4, and 7 after different epochs of training. Since L^2-norm based regularization is used for pretraining, the distribution of the weights at epoch zero is uni-modal with a single peak at zero. As training with SYMOG begins, two additional peaks arise at $\pm 2^{-f}$ since the weights are clipped to the quantization domain $\left[-2^{-f}, 2^{-f}\right]$ after each update step (clearly visible at epoch 20). With an increasing training time, the layer weights are continuously rearranged into a distribution consisting of three symmetric Gaussian modes. The variance of the Gaussian modes is continuously decreased by an exponentially increasing regularization parameter. After 100 epochs of training, the variance is so small that the remaining quantization error has no significant impact on the accuracy of the learning task.

On the other hand, Figure 4.3 visualizes the weight adaptation by showing the percentage of weights that switch to another fixed-point mode during a single epoch. The upper plot shows the result with weight clipping enabled as shown in Algorithm 1. Due to a small but exponentially increasing regularization parameter, the regularization effect is rather low at the beginning of the training, favoring an increased fluctuation of the weights. For example, on average 22% of the weights in the 7th layer change their fixed-point mode during each epoch in the first half of the training. After 80 epochs, when the variance of the Gaussian modes is already significantly smaller (see Figure 4.2), still 1.8% of the weights change their fixed-point mode until the end of training. However, the adaptation of the weights behaves differently from layer to layer. For example, the adaptation of the weights in the 1st layer is completed first after 32 epochs of training, whereas the weights in the 10th layer change their fixed-point mode until the end of the training. This may come from different step sizes and layer-dependent gradient scales, which results from the layer-wise mean squared error used in Equation 4.2.

In contrast, the lower plot in Figure 4.3 shows the percentage of weights that switch to another fixed-point mode with weight clipping turned off. First of all, it is noticeable that the number of changing modes is significantly lower for all layers. For example, on average 8% of the weights in the 7th layer change their fixed-point mode during each epoch in the first half of the training (compared to 22% when weight clipping is enabled). Furthermore, one can observe that the changing rates of the 4th and 7th layer increase once again after epoch 40. This is due to weights that are located outside the quantization domain $[-2^{-f}, 2^{-f}]$. These weights have to pass the distance to the outlying modes first.

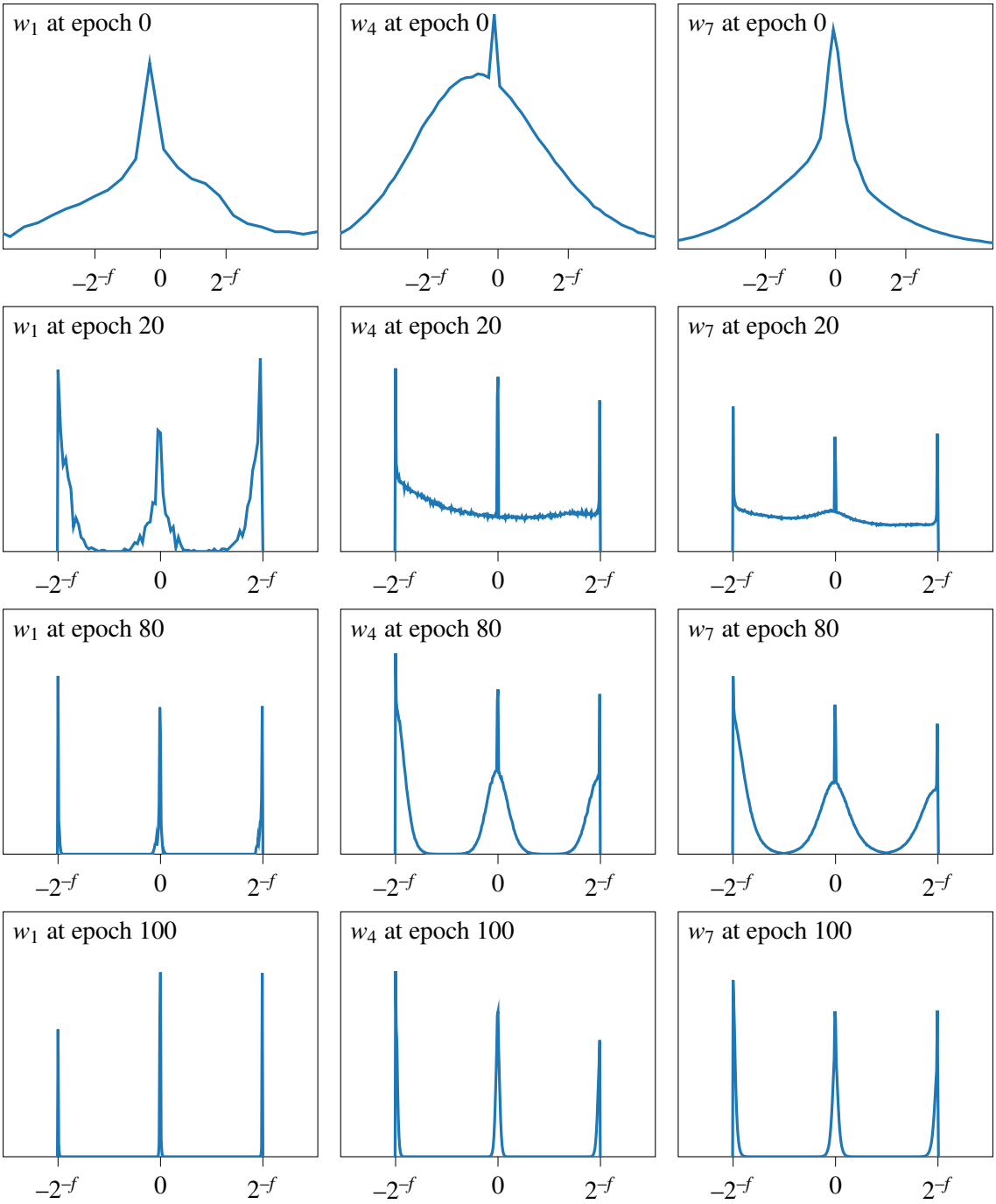

Figure 4.2: Weight distributions of the layers with index 1, 4, and 7 of VGG11 after different epochs of training. Since L^2-regularization is used for pretraining, the distribution of the weights at epoch zero is uni-modal with a single peak at zero. Then, training with SYMOG clips the weights to the domain $\left[-2^{-f}, 2^{-f}\right]$ and continuously rearranges them into a three-modal Gaussian distribution. The variance of each mode is continuously decreased as training progresses. After 100 epochs, the weights are that close to the fixed-point centers that post-quantization does not produce a remarkable quantization error. **Note**: the y-axes are scaled individually for convenience.

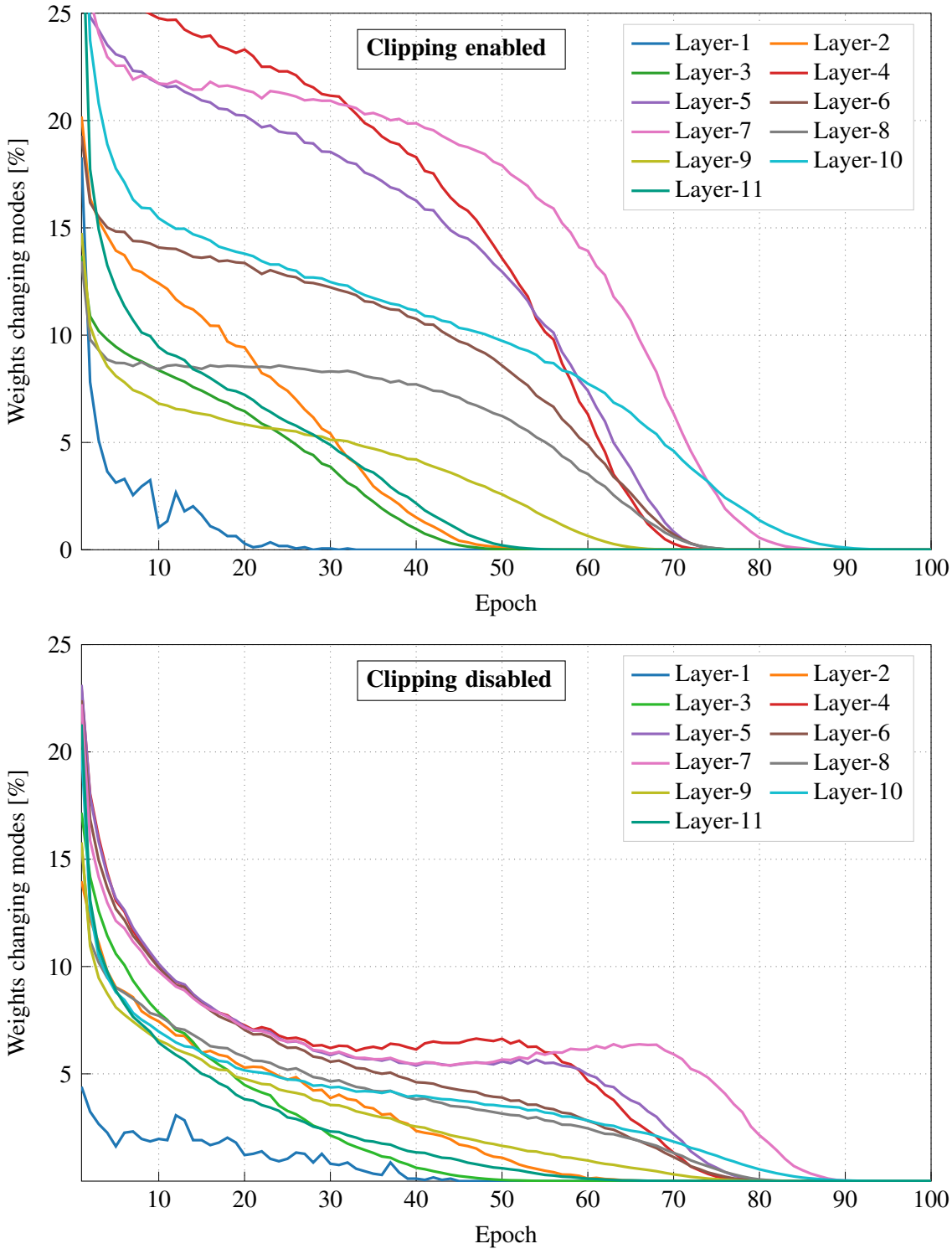

Figure 4.3: Both plots illustrate the weight adaptation in SYMOG training. The y-axes give the percentage of weights that change their fixed-point mode during a single epoch. The upper plot results if weight clipping is used as described in Algorithm 1, the lower plot results if the clipping is disabled. One can observe that the weight adaptation is improved by clipping the weights to the quantization domain. Especially in the very beginning of training, many weights are rearranged. Thus, the weight clipping improves SYMOG in both accuracy and training time.

4.3 EEquant: Easy and Effective Quantization of Deep Neural Networks

In this section, we propose EEquant, an **e**asy and **e**ffective **quant**ization approach to evaluate deep neural networks using pure fixed-point arithmetic after training. First, we extend our SYMOG soft quantization approach proposed in Section 4.2 by taking into account the parameters of the batch-normalization layers. In so doing, each batch-normalization layer can be folded into the preceding convolutional or fully-connected layer after training, resulting in a multi-modal distribution of the folded parameters well qualified for post-quantization. Furthermore, we use a discrete ReLU activation function to quantize the layer activations during each forward pass. Thus, the trained networks require significantly less memory and can furthermore be evaluated without the need for floating-point operations.

4.3.1 Reduction Loss

After training deep neural networks, the batch-normalization layers (if included) can be folded into the preceding convolutional or fully-connected layers as shown in Section 2.1.1, see Equation 2.9. This increases efficiency by reducing the number of both parameters and multiplications. However, to further improve efficiency through quantization, the weights of the folded layers must be quantized into a fixed-point representation after training. Therefore, the parameters of the convolutional or fully-connected layer with index l $\{w_l, b_l\}$ as well as the parameters of the corresponding batch-normalization layer $\{\gamma_l, \beta_l\}$ must be optimized during training to reduce both the learning loss and the expected quantization error of the folded parameters $\{\widehat{w}_l, \widehat{b}_l\}$. Here, $\{\widehat{w}_l, \widehat{b}_l\}$ are calculated according to Equation 2.9 (for a detailed description of the folding process as well as the symbols used, please see the paragraph about batch-normalization layers in Section 2.1.1).

In SYMOG, the expected quantization error is minimized by training deep neural networks using the L^2-norm based reduction loss from Equation 4.2. To transfer the same approach to the parameters of the folded layers, we introduce the following **weight regularization** term:

$$R_w = \sum_{l=1}^{L} \frac{1}{2} \left\| w_l \frac{\gamma_l}{\sqrt{\sigma_l^2 + \epsilon}} - Q^S \left(w_l \frac{\gamma_l}{\sqrt{\sigma_l^2 + \epsilon}}, f_{w,l}, B_w \right) \right\|_2^2 = \sum_{l=1}^{L} \frac{1}{2} \left\| \widehat{w}_l - \widetilde{w}_l \right\|_2^2 . \tag{4.5}$$

Here, Q^S is the signed fixed-point quantization function from Equation 2.23, $\widetilde{w}_l$ is the quantized version of the folded weight tensor $\widehat{w}_l$, $f_{w,l}$ is the position of the decimal point of the fixed-point representation of the weights in layer l, and B_w is the bit size of the quantized weights. Hence, the weight regularization penalizes the quantization error caused by folding the batch-normalization layers into the preceding convolutional or fully-

connected layers. The gradients of R_w with respect to the trainable parameters w_l and γ_l are as follows:

$$\frac{\partial R_w}{\partial w_l} = \left(\widehat{w}_l - \widetilde{w}_l\right)\left(\frac{\partial \widehat{w}_l}{\partial w_l} - \frac{\partial \widetilde{w}_l}{\partial w_l}\right) = \left(\widehat{w}_l - \widetilde{w}_l\right)\frac{\gamma_l}{\sqrt{\sigma_l^2 + \epsilon}}\,, \tag{4.6}$$

$$\frac{\partial R_w}{\partial \gamma_l} = \left(\widehat{w}_l - \widetilde{w}_l\right)\left(\frac{\partial \widehat{w}_l}{\partial \gamma_l} - \frac{\partial \widetilde{w}_l}{\partial \gamma_l}\right) = \left(\widehat{w}_l - \widetilde{w}_l\right)\frac{w_l}{\sqrt{\sigma_l^2 + \epsilon}}\,. \tag{4.7}$$

As in the case of our SYMOG approach, the derivative of the quantization function can be considered zero for any input value. Thus, both w_l and γ_l can be optimized during training such that the folded weights $\widehat{w}_l$ resemble a multi-modal distribution with minimal quantization error that can be quantized well after training.

In order to apply the same approach to the bias, we define the following **bias regularization** term:

$$\begin{aligned} R_b &= \sum_{l=1}^{L} \frac{1}{2} \left\| (b_l - \mu_l)\frac{\gamma_l}{\sqrt{\sigma_l^2 + \epsilon}} + \beta_l - Q^S\left((b_l - \mu_l)\frac{\gamma_l}{\sqrt{\sigma_l^2 + \epsilon}} + \beta_l,\, f_{b,l},\, B_b\right)\right\|_2^2 \\ &= \sum_{l=1}^{L} \frac{1}{2} \left\| \widehat{b}_l - \widetilde{b}_l \right\|_2^2 . \end{aligned} \tag{4.8}$$

Here, $\widetilde{b}_l$ is the quantized version of the folded bias $\widehat{b}_l$, $f_{b,l}$ is the position of the decimal point of the bias in layer l, and B_b is the bit size of the quantized bias. Consequently, the gradients of R_b with respect to the trainable parameters b_l, γ_l, and β_l are as follows:

$$\frac{\partial R_b}{\partial b_l} = \left(\widehat{b}_l - \widetilde{b}_l\right)\left(\frac{\partial \widehat{b}_l}{\partial b_l} - \frac{\partial \widetilde{b}_l}{\partial b_l}\right) = \left(\widehat{b}_l - \widetilde{b}_l\right)\frac{\gamma_l}{\sqrt{\sigma_l^2 + \epsilon}}\,, \tag{4.9}$$

$$\frac{\partial R_b}{\partial \gamma_l} = \left(\widehat{b}_l - \widetilde{b}_l\right)\left(\frac{\partial \widehat{b}_l}{\partial \gamma_l} - \frac{\partial \widetilde{b}_l}{\partial \gamma_l}\right) = \left(\widehat{b}_l - \widetilde{b}_l\right)\frac{b_l - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}}\,, \tag{4.10}$$

$$\frac{\partial R_b}{\partial \beta_l} = \left(\widehat{b}_l - \widetilde{b}_l\right)\,. \tag{4.11}$$

When calculating the gradients, it is also feasible to compute the derivatives with respect to the decimal points f_w and f_b. However, if EEquant reduces a pretrained model, we recommend initializing f_w and f_b with the integer values that provide the lowest initial quantization error on the pretrained weights.

Since both the weights and the bias each require a separate regularization term, we define the reduction loss as follows:

$$\mathcal{L}_{\text{reduce}} = R_w + R_b\,. \tag{4.12}$$

Consequently, the reduction loss penalizes the quantization errors that would result from quantizing the parameters of the folded layers. During training, the reduction loss is added to the learning loss $\mathcal{L}_{\text{learn}}$ according to

$$\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda \, \mathcal{L}_{\text{reduce}} \, . \tag{4.13}$$

Here, $\mathcal{L}_{\text{train}}$ is the overall training objective, and λ is the regularization parameter that controls the weighting between both losses. The larger λ, the higher the contribution of the reduction loss whose minimization reduces of the model capacity. As for our SYMOG approach, we initialize λ with a comparatively small start value to spend more model capacity at the beginning of the training. The regularization parameter is then exponentially increased over the training time according to:

$$\lambda(t) = \lambda_0 \exp\left(\alpha \frac{e}{E}\right) \, . \tag{4.14}$$

Here, λ_0 denotes the start value, α the exponential growth factor, t the current iteration, and T the total number of iterations. For example, if the training data consists of 1000 samples, the batch size is 10, and the number of training epochs is 5, the total number of iterations is 500.

4.3.2 Weight Clipping

As for our SYMOG approach, EEquant promotes posterior distributions of the parameters that are well-qualified for post-quantization. Consequently, each layer has a corresponding quantization domain within which the weight values should be located to avoid saturation effects. The procedure is illustrated in Figure 4.1. Taking into account the folding of the batch-normalization layers, the quantization domain of the weights in layer l is as follows:

$$\left[\frac{\sqrt{\sigma_l^2 + \epsilon}}{\gamma_l} \, 2^{-f_{w,l}} \, \left(-2^{B_w - 1}\right) , \; \frac{\sqrt{\sigma_l^2 + \epsilon}}{\gamma_l} \, 2^{-f_{w,l}} \, \left(2^{B_w - 1} - 1\right) \right] \, . \tag{4.15}$$

Here, both the parameters of the batch-normalization layers and the parameters of the fixed-point representation (i.e. the decimal point and the bit size) determine the respective quantization domain. Therefore, to speed up the training, the weights are clipped after each update step according to the quantization domain 4.15.

4.3.3 Fixed-Point Activations

The rectified linear unit (ReLU) is the state-of-the-art non-linear activation function in deep neural networks [3]. As mentioned in Section 2.1.1, ReLU activation layers are usually located subsequent to batch-normalization layers and set negative input values to

zero. Since the activations, unlike the parameters, cannot be reliably optimized to a multi-modal distribution using regularization, we quantize the ReLU activations during each forward pass. Therefore, we use the unsigned (i.e. non-negative) fixed-point quantization function from Equation 2.23 and combine it with the ReLU function from Equation 2.11. Accordingly, the quantized activation is implemented as follows:

$$\widetilde{x_l} = Q^U\left(\widehat{a_l}, f_{x,l}, B_x\right) = \text{clip}\left(\left\lfloor \frac{\widehat{a_l}}{2^{-f_{x,l}}} \right\rceil, 0, 2^{B_x} - 1\right) 2^{-f_{x,l}} . \tag{4.16}$$

Here, l is the layer index, $\widehat{a_l}$ the output of the preceding batch-normalization layer, $\widetilde{x_l}$ the quantized activation, $f_{x,l}$ the position of the decimal point of the quantized activations in layer l, and B_x the bit size of the quantized activations. Since the derivative of the rounding function is zero almost everywhere (see Figure 2.2), we use the straight-through estimator [53] during each backward pass to approximate its gradient as follows:

$$\frac{\partial \lfloor x \rceil}{\partial x} := 1 . \tag{4.17}$$

Thus, the gradient of the quantized activation $\widetilde{x_l}$ with respect to its input $\widehat{a_l}$ can be approximated as follows:

$$\frac{\partial \widetilde{x_l}}{\partial \widehat{a_l}} = \begin{cases} 0 & \text{if } \widehat{a_l} < 0 \\ 0 & \text{if } \widehat{a_l} > (2^{B_x} - 1)\, 2^{-f_{x,l}} \\ 1 & \text{if else.} \end{cases} \tag{4.18}$$

In this way, training with quantized activations is feasible. As in the case of the weight and bias regularization (Equation 4.5 and 4.8), the decimal points of the activation quantization are initialized with the integer values that provide the lowest quantization errors based on a pretrained model. Therefore, a subset of the training data is propagated through the network to minimize the resulting mean squared quantization error, i.e. min. $\|x_l - \widetilde{x_l}\|_2^2$ so that $f_{x,l} \in \mathbb{Z}$. For more details, please see also Algorithm 2.

4.3.4 Implementation Details

Algorithm 2 summarizes our EEquant approach to train a deep neural network that can be evaluated using pure fixed-point arithmetic after training. The pretrained model f_Θ, the number of training epochs E, the training set $D = \{(x_i, y_i)\}_{i=1}^d$, the batch size S, the learning loss $\mathcal{L}_{\text{learn}}$, the learning-rate domain $[\eta_1, \eta_E]$, the start value λ_0 and growth-factor α of the regularization parameter, as well as the bit sizes B_w and B_x are required as input values. For simplicity, we choose B_w and B_x according to the desired model complexity and define the bit size of the bias B_b as $B_b := 2B_x$. Regarding the learning rate and the regularization parameter, we use $[\eta_0, \eta_E] = [0.01, 0.001]$ and $\lambda_0 = 0.001$ for the CIFAR classification

Algorithm 2 EEquant: Training a deep neural network with both minimal quantization error and fixed-point activations. EEquant takes into account the distributions of the parameters of the batch-normalization layers. After training, the batch-normalization layers can be folded into the preceding convolutional or fully-connected layers, with the parameters of the folded layers resembling a multi-modal distribution.

Input: Pretrained model f_Θ, Number of Epochs E, Training Data $D = \{(x_i, y_i)\}_{i=1}^{d}$, Batch size S, Learning loss $\mathcal{L}_{\text{learn}}$, Learning-rate domain $[\eta_0, \eta_E]$, Regularization start value λ_0 and growth-factor α, Desired bit sizes B_w and B_x.
Replace all ReLU activations with Equation 4.16
for $l = 1$ to L **do**
    min. $\left\|\widehat{w}_l - Q^S\left(\widehat{w}_l, f_{w,l}, B_w\right)\right\|^2 + \left\|\widehat{b}_l - Q^S\left(\widehat{b}_l, f_{b,l}, 2B_x\right)\right\|^2 + \left\|x_l - Q^U\left(x_l, f_{x,l}, B_x\right)\right\|^2$
    s.t. $\left\{f_{w,l}, f_{b,l}, f_{x,l}\right\} \in Z$
end for
for $e = 1$ to E **do**
    $\eta \leftarrow \eta_0 - (\eta_0 - \eta_E)\, e/E$
    Randomly shuffle D.
    Divide D into N batches $\{X_n, Y_n\}_{n=1}^{N}$ of size S.
    for $n = 1$ to N **do**
        $\lambda \leftarrow \lambda_0 \cdot \exp\left(\alpha \frac{e\,n}{E\,N}\right)$
        $\widehat{Y}_n \leftarrow f_\Theta(X_n)$
        Compute $\mathcal{L}_{\text{learn}}\left(\widehat{Y}_n, Y_n\right)$
        Compute $\mathcal{L}_{\text{reduce}}$ according to Equation 4.12
        Compute $\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda\, \mathcal{L}_{\text{reduce}}$
        Compute $\frac{\partial \mathcal{L}_{\text{train}}}{\partial w}$
        $w \leftarrow \text{SGD}\left(w, \eta, \frac{\partial \mathcal{L}_{\text{train}}}{\partial w}\right)$
        for $l = 1$ to L **do**
            $w_l \leftarrow \text{clip}\left(w_l,\ 2^{-f_l}\left(-2^{B-1}\right),\ 2^{-f_l}\left(2^{B-1} - 1\right)\right)$
        end for
    end for
end for
for $l = 1$ to L **do**
    Compute $\widehat{w}_l$ and $\widehat{b}_l$ according to Equation 2.9
    $\widetilde{w}_l \leftarrow Q^S\left(\widehat{w}_l, f_{w,l}, B_w\right)$
    $\widetilde{b}_l \leftarrow Q^S\left(\widehat{b}_l, f_{b,l}, 2B_x\right)$
end for
return $\{\widetilde{w}_l, \widetilde{b}_l\}_{l=1}^{L}$

tasks, and $[\eta_0, \eta_E] = [0.0001, 0.00001]$ and $\lambda_0 = 0.01$ for the ImageNet classification task. Furthermore, we use $\alpha = 10$. However, these values can be adjusted according to the learning task. In general, the value of the regularization parameter should be sufficiently high to ensure that the reduction loss is close to (or equal to) zero at the end of training.

Scheduling and Preparation: Initially, all ReLU functions used in f_Φ are replaced with the fixed-point ReLU activation function proposed in Equation 4.16. Subsequently, the layer-wise decimal points of the weights, the bias, and the activations are initialized such that the corresponding quantization error is minimized based on the pretrained model f_Φ (see lines 3 to 6). Since this is a discrete optimization problem with a finite number of possible values, the solution can be found by a grid search. To find the layer-wise decimal points of the activations, a subset of the training data is propagated through the model. Furthermore, the bit sizes of the weights, the bias, and the activations are initialized.

Training with EEquant: Next, training with EEquant begins. Before each epoch, the learning rate is scheduled and linearly decreased from η_0 to η_E over the training epochs (see line 8). Furthermore, the training data is shuffled and divided into N batches of size S. For the CIFAR classification tasks, we use a batch size of 128. In the case of the ImageNet classification task, a batch size of 32 is used.

In EEquant, a single training step is implemented as follows. First, the regularization parameter is scheduled according to the current iteration (see line 12), and the respective mini-batch is propagated through the network to calculate its prediction (see line 13). After calculating the learning loss, the reduction loss is calculated according to Equation 4.12 and added to the learning loss (see lines 14 to 16). The resulting training loss is derived with respect to the model parameters before each parameter is updated in the direction of its negative gradient (see lines 17 to 18). Here, we use SGD optimization with the Nesterov momentum set to 0.9 [28]. After the update step, the layer weights are clipped according to the quantization domain 4.15 (see line 20). The procedure is repeated until the number of epochs is reached. Thus, EEquant fits seamlessly into the common training procedure of deep neural networks. Compared to hard quantization approaches, the batch-normalization layers do not have to be folded into the preceding layers during training.

Post Quantization: After the training with EEquant is completed, the batch-normalization layers are folded into the preceding layers according to Equation 2.9, and the parameters of the folded layers are quantized using the signed fixed-point quantization function Q^S from Equation 2.23. Thus, the final model runs without batch-normalization layers, and has all parameters and activations in fixed-point representation with per-tensor decimal points.

Bit sizes	SGD	Adam	RMSprop
4/8	**68.7**%	68.5%	68.1%
4/4	68.0%	**68.1**%	67.7%

Table 4.2: EEquant with VGG11 on CIFAR-100. Fixed-point accuracy after 3 epochs using SGD, Adam, and RMSprop. The baseline is 69.1%. The bit sizes are (weights/activations).

Bit sizes	SGD	Adam	RMSprop
4/8	91.4%	**91.7**%	90.2%
4/4	**91.1**%	90.8%	89.9%

Table 4.3: EEquant with ResNet-20 on CIFAR-10. Fixed-point accuracy after 3 epochs using SGD, Adam, and RMSprop. The baseline is 91.9%. The bit sizes are (weights/activations).

4.3.5 Experiments and Results

In this section, we evaluate EEquant on different benchmark data sets. First, we show invariance towards different optimizers on the classification tasks CIFAR-10 and CIFAR-100. Second, we illustrate the training procedure and show the negative correlation between the reduction loss and the accuracy of the corresponding fixed-point model. Subsequently, we show state-of-the-art performance on the ImageNet classification task. Therefore, we use different network architectures and compare our approach with recent fixed-point quantization approaches. Here, all models are initialized with pretrained weights. We do not use any data augmentation but normalize each input channel by subtracting the mean and dividing by the standard deviation over the training set. All experiments are done using Algorithm 2. The data sets and architectures used are described in detail in Section 3.

Invariance towards different optimizers: We claim that EEquant provides effective and stable optimization since high-precision parameters are used during training. Indeed, Chen *et al.* reported the convergence issues that may occur with hard quantization methods that quantize the parameters during each forward pass [54]. Their main observation: the training success depends most on the optimizer used. Especially between SGD and Adam, there are significant differences. Furthermore, an SGD based optimization fails to maintain the floating-point accuracy in many cases.

In order to show that EEquant is invariant towards different optimizers, we make experiments on the CIFAR classification tasks using SGD, Adam, and RMSprop optimization [27, 28]. The training time is three epochs, and we use a batch size of 128. All optimizers are initialized using their default values according to their PyTorch implementation [37]. We train ResNet-20 [5] on CIFAR-10 and VGG11 [11] on CIFAR-100, the results are shown in Table 4.2 and Table 4.3. For both classification tasks, all three optimizers provide similar fixed-point performances, with Adam and SGD being slightly superior to RMSprop. Furthermore, the fixed-point accuracies with 4-bit weights and 8-bit activations almost reach the baseline performances, which are based on 32-bit floating-point weights and

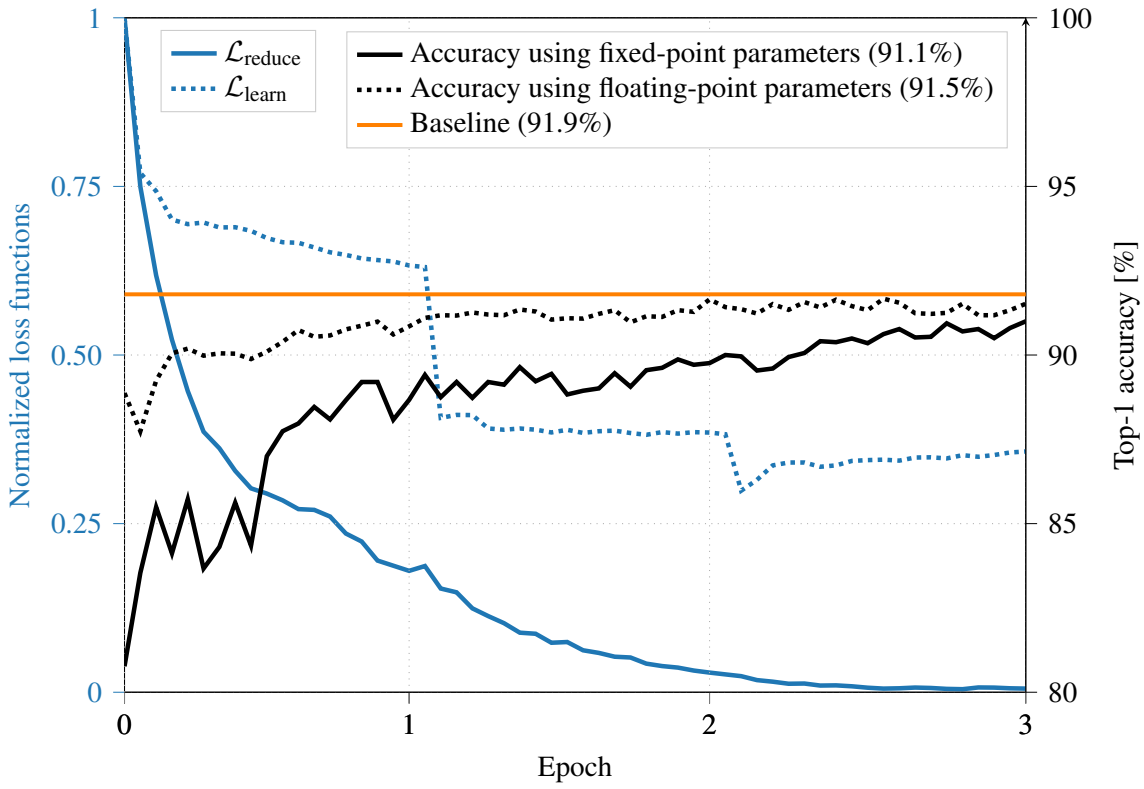

Figure 4.4: Training with EEquant using ResNet-20 on CIFAR-10. The bit sizes are 4 bits for both the weights and activations. The training loss consists of the learning loss $\mathcal{L}_{\text{learn}}$ and the reduction loss $\mathcal{L}_{\text{reduce}}$, which are both normalized on the left y-axis so that their maximum value is one. On the right y-axis, the test accuracies of both the floating-point model and the fixed-point model are shown. Here, the fixed-point accuracy indicates the test performance that would have been achieved if the parameters had been quantized after the respective training step (e.g. quantizing the parameters after one epoch would have resulted in a test accuracy of 88.7% compared to a floating-point accuracy of 91.1%). With EEquant, minimizing the learning loss increases the performance using floating-point parameters. Furthermore, minimizing the reduction loss decreases the performance gap obtained using floating-point parameters on the one hand and fixed-point parameters on the other.

activations.

Correlation between the reduction loss and the fixed-point accuracy: The approach of EEquant is to minimize both the learning loss $\mathcal{L}_{\text{learn}}$ and the reduction loss $\mathcal{L}_{\text{reduce}}$ simultaneously during training. While reducing the learning loss increases the accuracy of the model that uses floating-point parameters, we claim that minimizing the reduction loss sufficiently prepares the parameters for an accurate post-quantization. Consequently, the value of $\mathcal{L}_{\text{reduce}}$ should be correlated with the difference between the test accuracies

obtained using floating-point parameters on the one hand and fixed-point parameters on the other. To review our claim, Figure 4.4 compares the two losses and the test accuracies of ResNet-20 on CIFAR-10 using 4-bit weights and activations. Here, both the learning loss and the reduction loss are each normalized on the left y-axis so that their maximum value is one. The accuracies are shown on the right y-axis and indicate the test performances that would have been achieved using either floating-point parameters or their quantized counterparts at the current time step. ResNet-20 is initialized using pretrained weights with a corresponding baseline accuracy of 91.9%. Due to an initial learning rate of 0.01 (which is significantly higher than the final learning rate of the pretraining) and the inclusion of the reduction loss, the accuracy using floating-point parameters diverges from the baseline accuracy at the beginning of the training. However, with a decreasing learning loss, the accuracy increases again and approaches the baseline. Furthermore, as the reduction loss decreases, the difference between the accuracies obtained with either floating-point parameters or fixed-point parameters decreases. Thus, the trained model has both good performance and low quantization error.

Furthermore, Figure 4.5 illustrates the functionality of EEquant using the example of the 13th layer of ResNet-20. On the left side, the distribution of the weights w of the unfolded convolutional layer is shown after several epochs of training. On the right side, the corresponding distribution of the folded weights $\widehat{w}$ are shown ($\widehat{w}$ results from folding the corresponding batch-normalization layer into the convolutional layer according to Equation 2.9). Here, a bit size of 4 results in 16 fixed-point modes with uniform distance. As noticeable, the quantization of the unfolded weights w would lead to a significant quantization error. However, EEquant optimizes the trainable parameters of the convolutional layer $\{w, b\}$ as well as the batch-normalization layer $\{\gamma, \beta\}$ so that the folded weights $\{\widehat{w}, \widehat{b}\}$ resemble a multi-modal distribution with low quantization error. Thus, the folded weights can be quantized with no significant loss in accuracy after training.

ImageNet classification task: For the ImageNet classification task, we use MobileNetV1, MobileNetV2, InceptionV3, and ResNet-50. The training time is three epochs. We compare with the following quantization approaches: Trained Quantization Thresholds (TQT, [18]), Google's Quantization-Aware Training (QAT, [13, 50]), and Fast Adjustable Threshold for Uniform Neural Network Quantization (FAT, [55]). All methods use pretrained weights as initialization.

Besides the test accuracy of the quantized model, there are certain criteria to rate the capability of individual methods concerning their application on embedded devices: **symmetric** vs **asymmetric** quantization functions, **per-tensor** vs **per-channel** step-sizes, and **power-of-two** vs **real-valued** step-sizes. Symmetric quantization functions do not shift zero points and therefore avoid cross-terms within matrix multiplications, per-tensor step-sizes share the same value across all weights or activations of a tensor, and power-of-two step-sizes enable bit-shift operations. Thus, the most economical quantization function is

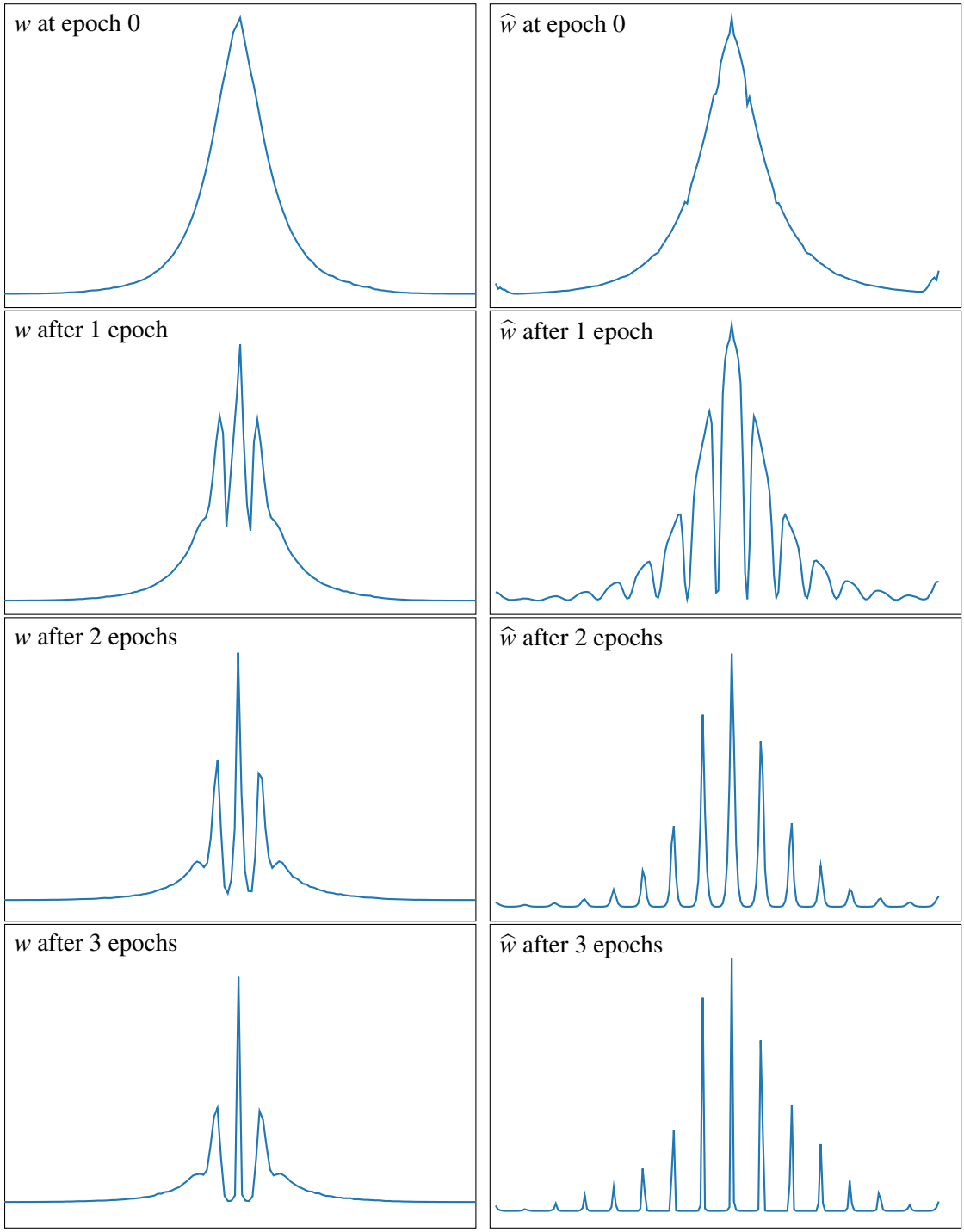

Figure 4.5: Qualitative illustration of the functionality of EEquant using the example of the 13th layer of ResNet-20. On the left side, the distribution of the weights w of the unfolded convolutional layer is shown after several epochs of training. On the right side, the corresponding distribution of the folded weights $\widehat{w}$ are shown ($\widehat{w}$ results from folding the corresponding batch-normalization layer into the convolutional layer according to Equation 2.9). Comparisons are made at different training times. The bit size is 4 bits, resulting in 16 fixed-point modes with uniform distance. As noticeable, the quantization of w would lead to a high quantization error. However, the goal is to optimize the trainable parameters $\{w, b, \gamma, \beta\}$ so that their folded counterparts $\{\widetilde{w}, \widetilde{b}\}$ yield a multi-modal distribution with a small quantization error.

MobileNetV1 on ImageNet

Table 4.4: Top-1 accuracies and quantization criteria of different fixed-point quantization methods using MobileNetV1 on ImageNet. The best quantization function to implement is symmetric and uses per-tensor scaling factors that are powers-of-two. ○ = False, ● = True.

Method	Bit sizes (w/x)	per-tensor	symmetric	pow-of-2	Top-1 Accuracy
Baseline	32 / 32	○	○	○	70.9%
QAT [13]	8 / 8	○	●	○	70.7%
QAT [13]	8 / 8	●	○	○	70.0%
TQT [18]	8 / 8	●	●	●	**71.1**%
EEquant	8 / 8	●	●	●	71.0%

MobileNetV2 on ImageNet

Table 4.5: Top-1 accuracies and quantization criteria of different fixed-point quantization methods using MobileNetV2 on ImageNet. The best quantization function to implement is symmetric and uses per-tensor scaling factors that are powers-of-two. ○ = False, ● = True.

Method	Bit sizes (w/x)	per-tensor	symmetric	pow-of-2	Top-1 Accuracy
Baseline	32 / 32	○	○	○	71.9%
FAT [55]	8 / 8	○	●	○	71.1%
FAT [55]	8 / 8	●	○	○	19.9%
FAT [55]	8 / 8	●	●	○	8.1%
QAT [13]	8 / 8	○	●	○	71.1%
QAT [13]	8 / 8	●	○	○	70.9%
TQT [18]	8 / 8	●	●	●	**71.8**%
EEquant	8 / 8	●	●	●	71.4%

InceptionV3 on ImageNet

Table 4.6: Top-1 accuracies and quantization criteria of different fixed-point quantization methods using InceptionV3 on ImageNet. The best quantization function to implement is symmetric and uses per-tensor scaling factors that are powers-of-two. ○ = False, ● = True.

Method	Bit sizes (w/x)	per-tensor	symmetric	pow-of-2	Top-1 Accuracy
Baseline	32 / 32	○	○	○	78.0%
QAT [13]	8 / 8	●	○	○	75.4%
TQT [18]	8 / 8	●	●	●	**78.3**%
EEquant	8 / 8	●	●	●	78.1%
QAT [13]	7 / 7	●	○	○	75.0%
EEquant	7 / 7	●	●	●	**77.1**%
EEquant	6 / 6	●	●	●	76.2%

ResNet-50 on ImageNet

Table 4.7: Top-1 accuracies and quantization criteria of different fixed-point quantization methods using ResNet-50 on ImageNet. The best quantization function to implement is symmetric and uses per-tensor scaling factors that are powers-of-two. ○ = False, ● = True.

Method	Bit sizes (w/x)	per-tensor	symmetric	pow-of-2	Top-1 Accuracy
Baseline	32 / 32	○	○	○	75.2%
QAT [13]	8 / 8	●	○	○	74.9%
TQT [18]	8 / 8	●	●	●	75.4%
EEquant	8 / 8	●	●	●	**75.6**%
QAT [13]	4 / 8	○	○	○	73.2%
TQT [18]	4 / 8	●	●	●	**74.4**%
EEquant	4 / 8	●	●	●	73.4%

symmetric with per-tensor and power-of-two step-sizes.

MobileNets: The results are shown in Table 4.4 and Table 4.5. Due to their limited capacity and depth-wise separable convolutions, MobileNet architectures are considered difficult to quantize [18, 55]. However, with a performance of 71%, we can quantize MobileNetV1 to 8-bit fixed-point weights and activations with no loss in accuracy (71% vs 70.9% baseline accuracy). Thus, we achieve a similar performance compared to TQT and slightly better than QAT. On MobileNetV2, TQT provides the best result with a fixed-point accuracy of 71.8% compared to 71.4% of EEquant. Furthermore, both QAT and FAT are not capable of fulfilling more than one quantization criterion in a single experiment without a significant loss in accuracy. On the one hand, the accuracy of FAT drops below 10% when both per-tensor scaling factors and symmetric quantization functions are combined. On the other hand, QAT does not provide any experiments at all with two criteria fulfilled.

InceptionV3 and ResNet-50: The results of InceptionV3 are shown in Table 4.6. With 8-bit fixed-point weights and activations, our EEquant approach achieves a Top1 accuracy of 78.1%, which is slightly higher than the baseline accuracy of 78%. Here, TQT performs best with a very slight margin to EEquant. For 7-bit fixed-point weights and activations, our approach achieves the highest performance in comparison and outperforms QAT by more than 2%. Furthermore, we are the only ones that quantize InceptionV3 with 6-bit fixed-point weights and activations. Here, our approach achieves a Top1 accuracy of 76.2%, which is only 1.8% below the baseline accuracy.

The results of ResNet-50 are shown in Table 4.7. With 8-bit fixed-point weights and activations, our EEquant approach achieves a Top-1 accuracy of 75.6%, which is slightly better than the baseline accuracy and the performance reported by TQT. This may be due to regularization effect of the reduction loss. Using 4-bit weights and 8-bit activations, TQT yields a fixed-point accuracy of 74.5% compared to EEquant with 73.4%.

4.4 Related Work

Network quantization reduces the precision of the weights and activations by reducing their respective bit sizes. There are different quantization approaches that can be divided into three categories: post-training quantization, hard quantization, and soft quantization [12, 38]. The latter two coincide under the term quantization-aware training methods. An overview can also be seen in Figure 1.1.

4.4.1 Post-training Quantization

Post-training quantization transforms an already trained floating-point model into a quantized representation by analyzing the network statistics. The training of the floating-point model does not include any quantization constraints. Instead, the distributions of the weights (and optionally those of the layer activations) are analyzed in post-processing to find appropriate quantization functions for each layer and to correct the resulting quantization noise.

In [48], Lin *et al.* collected and analyzed statistics of the parameters and activations to estimate suitable quantization functions for each layer. Furthermore, Lin *et al.* enabled flexible bit sizes and proposed a method to find an optimal bit size assignment for each layer across the network. In [56], Choukroun *et al.* formulated the quantization of both the weights and activations as minimum mean squared error problem. In order to fine-tune the scaling factors of the linear quantization functions, Choukroun *et al.* utilized a small subset of the training data. In [57], Meller *et al.* combined factorization and quantization to reduce the degradation caused by weight quantization. On the one hand, Meller *et al.* rearranged the weights to make them less sensitive towards the quantization noise. On the other hand, they proposed transformations to align the ranges of the channels that belong to the same layer, which also makes the layer more robust to quantization. In [58], Banner *et al.* introduced a post-training quantization approach to quantize both the weights and activations to 4-bit. Therefore, Banner *et al.* proposed three approaches for minimizing the quantization error: First, approximating the optimal clipping value by minimizing the mean squared quantization error of the activations, second, allocating flexible bit sizes for all layer weights regarding their dynamic range, and third, updating the bias to correct the shift in the mean and the variance of the weights caused by quantization. Recently, Li *et al.* proposed a post-training quantization approach that is capable of reducing the bit sizes down to INT2 [59]. By using second-order analysis, Li *et al.* defined reconstruction units that compensate (at least in part) the quantization error.

In general, post-training quantization has both advantages and disadvantages. On the one hand, post-processing is not as time consuming as retraining the entire model using large amounts of the training data. On the other hand, low-bit configurations reduce accuracy tremendously in many cases.

4.4.2 Hard Quantization

Hard quantization methods quantize the weights and optionally also the activations during the training of deep neural networks. Nevertheless, to enable convergence, the local gradients of the quantization functions used must be estimated during the backward pass. Thus, hard quantization methods integrate the quantization noise into the training of deep neural networks and adapt the network parameters accordingly.

On the one hand, the hard quantization approach became popular for quantizing the weights with BinaryConnect, where Courbariaux *et al.* quantized the weights of the fully-connected layers to ± 1 during each forward pass [46]. During the backward pass, the gradients of the quantized weights were passed through to their high-precision counterparts. Practically, this was a variation of the straight-through estimator [53], which approximates the local gradient of the rounding function with one. In the following, Li and Liu increased the model capacity by combining ternary-valued weights with a real-valued scaling factor [36]. Here, an optimal value of the scaling factor was found by minimizing the euclidean distance between the high-precision weights and the scaled ternary-valued weights. Furthermore, Zha *et al.* investigated asymmetrical ternary weights with two independent scaling factors [49].

On the other hand, Courbariaux *et al.* introduced Binarized Neural Networks and quantized both the weights and activations to ± 1 during each forward pass [47]. During the backward pass, Courbariaux *et al.* utilized the straight-through estimator to approximate the derivative of the rounding function with one. However, parts of the network, such as the batch-normalization layers, remained in floating-point precision, which complicated an efficient implementation on dedicated hardware. As a result, the focus shifted towards embedded hardware constraints. The following steps essentially improved the usability on embedded devices: uniform quantization functions to enable variable integer domains [13, 55], symmetric and uniform quantization functions to restrict zero-points to 0 [13, 50, 55], per-tensor quantization to share the same step-size across all weights or activations of a single layer [13, 18], and power-of-two step-sizes to enable bit-shift operations [18, 44]. However, in order to merge the batch-normalization layers into the fixed-point precision of the preceding convolutional or fully-connected layers, the training graph has to be changed to first calculate the batch statistics of the layer activations and then the quantized output of the folded layers [13]. This increases the computation costs during training and the quantized weights start to jitter [50]. In the following, several modifications were presented to increase both the stability and the performance [50]. This includes, among others, corrections terms within the folded layers, as well as frozen batch statistics [50]. Furthermore, [18] froze all batch-normalization layers after a certain amount of iterations. These modifications further increased the implementation effort and introduce additional heuristics (e.g. finding a suitable time step for freezing the batch-normalization layers).

4.4.3 Soft Quantization

Although hard quantization methods achieve convergence, they induce a gradient mismatch since the derivative of the quantization functions used must be estimated during each backward pass. In order to avoid such gradient noise, soft quantization approaches use floating-point parameters during both the forward pass and the backward pass but simultaneously promote parameter distributions that are well-qualified for post-quantization.

In Figure 4.1, a simplified comparison between a uni-modal and a multi-modal weight distribution is given. For quantizing the weights using ternary values $\{-2^{-f}, 0, 2^{-f}\}$, the multi-modal weight distribution yields significantly less quantization error.

On the one hand, Bayesian methods were used for compressing deep neural networks using either low-bit weights or zero weights. These methods derive prior distributions of the weights that result in either sparse or multi-modal weight distributions. In [60], Ullrich *et al.* fitted a mixture of Gaussian priors over the model parameters to rearrange the weights of single layers around certain cluster centers. After the training, Ullrich *et al.* achieved compression by assigning each weight to one of these clusters. In [61], Louizos *et al.* proposed a prior to induce group sparsity within the network layers. In order to find an appropriate bit size for each layer to quantize its parameters using fixed-point precision, Louizos *et al.* analyzed the uncertainties of the posterior distributions. Furthermore, Achterhold *et al.* introduced a quantizing prior to train deep neural networks with multi-modal weight distributions that can be quantized using symmetric ternary values [38]. In addition, Achterhold pruned weights with high variance.

On the other hand, optimization strategies for ordinary deep neural networks (i.e. deterministic after the training) have been developed that progressively reduce the quantization error during training. In [62], Zhou *et al.* proposed a loss-error-aware training method for deep neural networks with low-bit weights. Zhou *et al.* repeatedly divided the weights into two partitions, constantly quantizing one part and retraining the second part using an additional regularization term. In [63], Choi *et al.* investigated both quantized weights and quantized activations using regularization. However, [62, 63] also kept floating-point batch-normalization layers as well as real-valued quantization step-sizes, which makes a complete fixed-point implementation of the network impossible.

4.5 Conclusion

In this chapter, we proposed two fixed-point quantization approaches for deep neural networks. Our methods use floating-point parameters during training but promote posterior distributions of the parameters that are well-qualified for post quantization. This is done by using an additional reduction loss during training. The reduction loss represents the respective fixed-point constraints and can be added to the learning loss for solving both the learning task and the fixed-point constraints simultaneously during training.

At first, we proposed SYMOG, an approach to train deep neural networks with minimal quantization error. During training, the distribution of the weights changes from a uni-modal to a symmetric and multi-modal distribution, with each mode corresponding to a certain fixed-point number. This allows the weights to be quantized with no significant loss in accuracy after training. Compared to hard quantization approaches, the computational graph remains unchanged and no gradient estimators are needed. Instead,

only the reduction loss has to be integrated into the training. Thus, SYMOG is easy to implement and we believe it is an appropriate quantization approach for a wide range of deep neural network applications. Furthermore, SYMOG enables quantizing the weights using symmetric ternary values with power-of-two scaling factors. Thus, most of the multiply-and-accumulate operations of the forward pass can be replaced by additions and subtractions. In multiple experiments we demonstrated the benefit of SYMOG in terms of training time and test accuracy: with the lowest number of training epochs, we achieved new state-of-the-art performance on MNIST, CIFAR-10, and CIFAR-100. Furthermore, we provided insights into the training and illustrated the effect of single components such as the weight clipping.

In addition, we proposed EEquant, a quantization approach to evaluate deep neural networks using pure fixed-point arithmetic. EEquant is the first soft quantization approach that considers the distribution of the parameters of the batch-normalization layers. As in the case of SYMOG, all parameters remain in floating-point precision during training. However, the training promotes distributions of the parameters that resemble a multi-modal distribution after folding the batch-normalization layers into the preceding convolutional or fully-connected layers. Compared to hard quantization approaches that require the batch-normalization layers to be folded during each training step, EEquant only needs to do this once after training, which makes training more efficient. Thus, EEquant has a comparatively low implementation effort. Furthermore, EEquant uses a discrete ReLU activation function that quantizes the layer activations using fixed-point arithmetic during each forward pass. Thus, deep neural networks trained with EEquant can be evaluated using pure fixed-point arithmetic after training. In various experiments on CIFAR-10 and ImageNet, EEquant reaches performance that is comparable to state-of-the-art hard quantization approaches.

Chapter 5

Network Pruning

Deep neural networks are usually overparameterized before training begins to increase the likelihood of getting adequate initial weights by random initialization. Consequently, trained neural networks have many redundancies and are computationally complex in terms of the number of parameters and required multiplications. In order to reduce complexity and improve their ability to generalize, redundant network connections can be pruned from the model architecture. Here, structured sparsity, as achieved by filter pruning, directly reduces the tensor sizes of the weights and activations and is thus particularly effective for reducing both the memory and the computational effort.

In this chapter, we propose Holistic Filter Pruning, a training procedure to reduce the memory and computational complexity of a deep neural network to a given target size. The user determines the target size in terms of the number of parameters and multiplications that are available on the target device. Training with Holistic Filter Pruning includes a reduction loss that calculates the difference between the actual model size and the target size. The reduction loss can be minimized during training by inducing sparsity over the channel-wise affine transformations of the batch-normalization layers. Thus, a global solution can be found that allocates the resources available over the individual layers such that the desired target size is fulfilled. In various experiments, we give insights into the training and achieve state-of-the-art performance on the CIFAR-10 and ImageNet classification tasks.

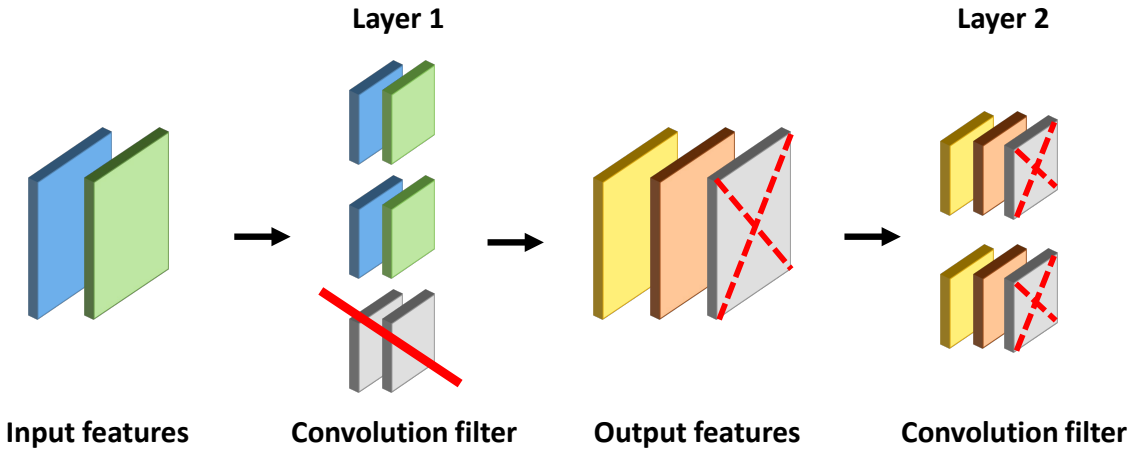

Figure 5.1: In order to reduce the complexity of deep neural networks, pruning methods reduce the number of parameters and multiplication by setting weight values to zero. However, pruning single weights leads to unstructured sparsity, which has only a minor impact on the memory requirements. In contrast, pruning entire filters and neurons results in a structured sparsity: Since filter pruning in Layer 1 reduces the number of filters as well as the number of output feature maps, the tensor sizes of both the weights and activations decrease. Furthermore, with a reduced number of output feature maps, the depth of the following Layer 2 decreases to the same degree.

5.1 Introduction

Deep neural networks have a strong ability for data abstraction and outperform classical methods in many machine learning challenges such as computer vision, object detection, or speech recognition [2, 42, 43]. But, recent progress has been made by training powerful models with many parameters using large-scale data sets [5, 11, 12]. Frankle and Carbin demonstrated the correlation between the initial model size and the probability of getting meaningful initial values for the parameters by random initialization: The chance that a subset of the parameters is initialized with values that constitute a suitable starting point for training increases with the number of parameters initialized with random values [10]. As a result, trained deep neural networks are usually over-parameterized, have high memory requirements, and need many floating-point multiplications, which are especially expensive concerning computation time and energy consumption [12].

However, reduction techniques can significantly reduce the complexity of trained deep neural networks. On the one hand, quantization reduces the precision of both the parameters and activations to accelerate neural networks on dedicated hardware [12, 44, 64]. On the other hand, pruning and factorization methods reduce the number of parameters and multiplications rather than their bit sizes [12, 65, 66]. structured sparsity, as achieved by filter pruning, reduces the computation time, energy consumption, and memory requirements of deep neural networks without the need for specialized hardware. A visualization of filter pruning is given in Figure 5.1.

Unsupervised filter pruning usually fails to preserve the accuracy of the original model without retraining. Therefore, data-driven approaches have been developed which either iteratively prune filters based on saliency scores [24, 65, 67, 68, 69, 70], or retrain the model under consideration of sparsity constraints [71, 72, 73, 74, 75].

Methods of the first category calculate saliency scores to rate the importance of individual filters. Such saliency scores can either be based on the magnitudes of the weights and activations [65, 67, 68], or on the discriminative power of single filters [69, 70]. Channels associated with low saliency scores are considered unimportant for fulfilling the learning task and are therefore deleted whereas the remaining filters are retrained. This process is repeated until the desired pruning rate is reached. However, determining saliency scores requires a lot of human labor and is usually a heuristic practice. Furthermore, layer-by-layer pruning, as well as iterative pruning and retraining, are unsuitable procedures for determining a global selection of filters to be pruned. Considering that all networks layers jointly contribute to the learning task, it is inappropriate to prune single layers independently. Moreover, iterative pruning and retraining may prune filters that become important again at a later iteration.

Methods of the second category investigate sparsity constraints that can be integrated into the training of deep neural networks. Many approaches apply regularization terms to push the sum of absolute values of filter weights or feature maps towards zero. Frequently used regularization terms are based on the L^0-norm [73], the L^1-norm [71], or on the LASSO regression [72]. Furthermore, [74, 75] introduced gate variables that scale single weights [74] or complete filters [75] by one or zero. In some cases, however, batch-normalization layers are neglected which reactivates previously pruned channels [75]. Furthermore, it is usually unfeasible to specify accurate pruning rates for the number of parameters and multiplications, which is why an iterative procedure of pruning and retraining becomes necessary.

In this chapter, we make the following contributions:

- We propose a holistic approach for reducing the number of parameters and required multiplication of a deep neural network by filter pruning. Therefore, the user specifies the desired model size in terms of the number of parameters and multiplications that are available on the target device. During training, a reduction loss is minimized that indicates the difference between the desired and the actual model size.

- The proposed method induces sparsity via the channel-wise scaling factors of the batch-normalization layers. Hence, no additional variables are needed. Furthermore, the pruning budget is allocated over the individual layers automatically such that the target size is reached. Thus, a global solution is found.

- We evaluate our pruning approach on two benchmark data sets. We provide comparisons with recent filter pruning results and prove state-of-the-art performance with

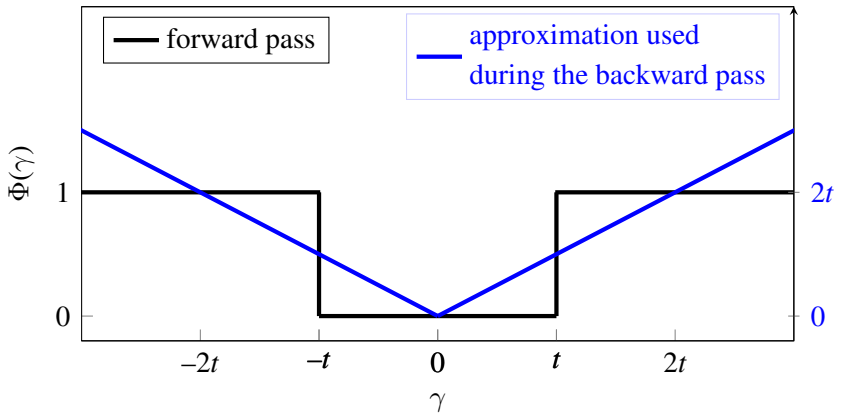

Figure 5.2: An illustration of the indicator function during both the forward and backward pass. During the forward pass, the indicator function outputs whether the absolute values of the batch-normalization scaling factors are greater than t. During the backward pass, the indicator function is approximated using two piece-wise linear functions. Thus, the gradient with respect to the scaling factor is either 1 or –1, depending on the sign of the scaling factor.

various network architectures. Furthermore, we analyze the allocation of pruning rates over the individual layers for different target sizes and layer types.

5.2 Technical Approach: Holistic Filter Pruning

First of all, this section introduces an indicator function that can be applied to the channel-wise scaling factors of the batch-normalization layers to distinguish between active and inactive channels. Next, a reduction loss is introduced that uses the indicator function to determine the difference between the desired model size and the actual model size in terms of the number of parameters and multiplication. Thus, minimizing both the learning loss and the reduction loss during training prunes channels across all layers such that the target size is fulfilled.

5.2.1 Indicator Function

According to Section 2.1.1, batch-normalization layers first normalize and then linearly transform each channel of a given input variable. Since they are furthermore applied after convolutional or fully-connected layers, the pruning of complete filters or neurons can be done via the channel-wise parameters of the batch-normalization layers: as the absolute values decrease, $\{\gamma_{l,c}, \beta_{l,c}\}$ scale the output of channel c in layer l towards zero. Here, $\gamma_{l,c}$ is the scaling parameter of the affine transformation, and $\beta_{l,c}$ is the corresponding bias (for more details, please see Section 2.1.1). Therefore, we first implement a magnitude-based indicator function that determines whether the absolute value of γ is smaller than the

magnitude t:

$$\Phi(\gamma,\, t) = \begin{cases} 0 & \text{if } |\gamma| \leq t \\ 1 & \text{if } |\gamma| > t \end{cases} . \tag{5.1}$$

If the indicator function outputs zero, the respective channel is considered negligible, since only the bias of the affine transformation remains (more on this below).

As can be seen in Figure 5.2, the indicator function is a non-smooth quantization function whose gradient is zero almost everywhere. Therefore, we utilize the straight-through estimator [53], which is widely used in network quantization to approximate the local gradient of the rounding function during the backward pass [18]. However, since the indicator function is symmetrical to the y-axis (in contrast, fixed-point quantization functions are usually symmetrical to the origin), we flip the estimator on the y-axis as well:

$$\frac{\partial \Phi(\gamma)}{\partial \gamma} := \begin{cases} -1 \text{ if } \gamma \leq 0 \\ 1 \text{ if } \gamma > 0 \end{cases} . \tag{5.2}$$

As shown in Figure 5.2, this is the most suitable approach for approximating the indicator function with linear segments. As a result, the gradient estimator is easy to implement and non-zero for all input values.

Liu *et al.* found that scaling factors with absolute values below 10^{-4} can be set to zero without a noticeable drop in accuracy [76]. Therefore, we use $t = 10^{-4}$ for our experiments. According to Equation 2.9, this results in the channel output being approximately equal to the batch-normalization bias $\beta_{l,c}$:

$$\widehat{a}_{l,c} = w_{l,c} \frac{\gamma_{l,c}}{\sqrt{\sigma_{l,c}^2 + \epsilon}} * x_{l-1,c} + (b_{l,c} - \mu_{l,c}) \frac{\gamma_{l,c}}{\sqrt{\sigma_{l,c}^2 + \epsilon}} + \beta_{l,c} \overset{|\gamma_{l,c}| < 10^{-4}}{\approx} \beta_{l,c} . \tag{5.3}$$

Here, $\{w_{l,c}, b_{l,c}\}$ are the weights and bias of channel c in layer l (which is either a convolutional or a fully-connected layer), $\{\mu_{l,c}, \sigma_{l,c}^2\}$ are the running estimates of the channel mean and variance, $\{\gamma_{l,c}, \beta_{l,c}\}$ are the learnable parameters of the corresponding batch-normalization layer, and $\widehat{a}_{l,c}$ is the batch-normalization output. The remaining batch-normalization bias $\beta_{l,c}$ is independent of the channel input. When propagating it through the following convolution or fully-connected layer, it shifts the resulting feature maps. However, this shift is corrected by the subsequent batch-normalization layer, which subtracts the mean over the respective mini-batch. After training, both the scaling factor γ and the bias β of the batch-normalization layers are set to zero if the indicator function outputs zero. Subsequently, the remaining filters and neurons are retrained to adjust the batch statistics.

If the network architecture used does not include batch-normalization layers, the indicator function can also be applied on learnable affine transformations that follow each convolution or fully-connected layer.

5.2.2 Reduction Loss

The complexity of a deep neural network results on the one hand from the number of its parameters P and on the other hand from the number of floating-point multiplications M that are needed to propagate one sample through the network. Furthermore, if P^* and M^* denote the number of parameters and multiplications that are available on the target device, the deviation between the actual model size and the target size can be described by the following reduction loss:

$$\mathcal{L}_{\text{reduce}} = \text{relu}\left(\frac{P - P^*}{P^0}\right) + \text{relu}\left(\frac{M - M^*}{M^0}\right) . \tag{5.4}$$

Here, P^0 and M^0 denote the number of parameters and multiplications of the original model. Thus, the terms within the rectifier functions denote the normalized differences between the actual model size (i.e. during training) and the target size. Using another notation, $1 - P/P^0$ denotes the actual pruning rate, whereas $1 - P^*/P^0$ denotes the desired pruning rate regarding the number of parameters. The rectifier functions are used to limit the reduction loss at zero whenever the desired pruning rates are reached. In contrast to the mean squared error, the rectifier function has the advantage of only penalizing deviations greater than zero. Consequently, both summands vary between zero and the desired pruning rates. For example, if the goal is to prune 50% of the parameters and 40% of the required multiplications, the reduction loss takes values between 0 and 0.9.

Both the original model size $\{P^0, M^0\}$ and the target sizes $\{P^*, M^*\}$ are constant values: the former is fixed whereas the latter is specified by the user according to the target device. Therefore, P and M remain the only variable quantities in Equation 5.4, which can be reduced by pruning filters and neurons from the network architecture. Utilizing the indicator function from Equation 5.1, the number of parameters in a feed-forward neural network can be calculated as follows:

$$P = \sum_{l=1}^{L} K_l^2 \, C_{l-1} \, C_l = \sum_{l=1}^{L} K_l^2 \, \left\|\Phi\left(\gamma_{l-1}\right)\right\|_1 \, \left\|\Phi\left(\gamma_l\right)\right\|_1 . \tag{5.5}$$

Here, l denotes the layer index, L the number of layers, C_l the number of active channels in layer l, and K_l is the kernel size of layer l (if layer l is fully-connected, K_l is equal to one). Here, we assume quadratic kernel tensors. Of course, it is also feasible to use non-quadratic kernel sizes. The number of active channels C_l can be calculated by accumulating over the output of the indicator function $\Phi\left(\gamma_l\right)$ (that is, a channel is considered active if the indicator function outputs 1 for the corresponding scaling factor). Since the indicator function outputs only non-negative values, the $p = 1$-norm can be used to calculate the sum over $\Phi\left(\gamma_l\right)$.

Calculating the gradient of P with respect to the scaling factor $\gamma_{l,c}$ results in:

$$\frac{\partial P}{\partial \gamma_{l,c}} = K_l^2 \left\| \Phi\left(\gamma_{l-1}\right) \right\|_1 \frac{\partial \Phi\left(\gamma_{l,c}\right)}{\partial \gamma_{l,c}} = \begin{cases} -K_l^2 \left\| \Phi\left(\gamma_{l-1}\right) \right\|_1 & \text{if } \gamma_{l,c} \leq 0 \\ K_l^2 \left\| \Phi\left(\gamma_{l-1}\right) \right\|_1 & \text{if } \gamma_{l,c} > 0 \end{cases} . \tag{5.6}$$

Hence, the gradient consists of the sign of the scaling factor and the respective channel size that is the number of parameters. Therefore, the magnitude of the gradient is proportional to the impact that the corresponding channel has on the model complexity (the larger the channel, the more its deletion reduces complexity). This is more useful than applying the L^1-norm to the batch-normalization scaling factors as recommended in [76]. The L^1 regularization does not consider the channel sizes and thus treats all channels equally in terms of complexity.

The same procedure can be done for calculating the number of multiplications:

$$M = \sum_{l=1}^{L} K_l^2 \, W_l \, H_l \, C_{l-1} C_l = \sum_{l=1}^{L} K_l^2 \, W_l \, H_l \, \left\| \Phi\left(\gamma_{l-1}\right) \right\|_1 \, \left\| \Phi\left(\gamma_l\right) \right\|_1 . \tag{5.7}$$

Here, W_l and H_l are the width and height of the output feature maps of layer l (see also Figure 2.1). Furthermore, the gradient of M with respect to $\gamma_{l,c}$ is as follows:

$$\frac{\partial M}{\partial \gamma_{l,c}} = \begin{cases} -K_l^2 \, W_l \, H_l \left\| \Phi\left(\gamma_{l-1}\right) \right\|_1 & \text{if } \gamma_{l,c} \leq 0 \\ K_l^2 \, W_l \, H_l \left\| \Phi\left(\gamma_{l-1}\right) \right\|_1 & \text{if } \gamma_{l,c} > 0 \end{cases} . \tag{5.8}$$

Consequently, after each forward pass, the reduction loss calculates the deviation between the current model size and the target size in terms of the number of parameters and required multiplications. The reduction loss can be minimized by pruning complete filters and neurons via the channel-wise scaling factors of the batch-normalization layers. During training, it is added to the learning loss $\mathcal{L}_{\text{learn}}$ according to

$$\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda \, \mathcal{L}_{\text{reduce}} . \tag{5.9}$$

Here, $\mathcal{L}_{\text{train}}$ is the overall training objective, and λ is the regularization parameter that controls the weighting between both losses: the larger λ, the higher the contribution of the reduction loss whose minimization results in a reduction of the model capacity. Thus, one intuitive approach is to scale both losses to the same magnitude. The same procedure for combining different loss functions has also been used in [77]. Therefore, we initialize λ such that $\lambda \, \mathcal{L}_{\text{reduce}}$ is equal to the expectation value of the learning loss over the training set. For example, if the average cross-entropy loss for an untrained model is 7.25 on the ImageNet classification task, and the desired pruning rate is 0.5 for both the parameters and the multiplications, λ is equal to 7.25. Furthermore, since we use pretrained models, we recommend heating up the pruning parameter over the training epochs.

5.2.3 Shortcut Connections

State-of-the-art deep neural networks such as ResNet or DenseNet [5, 35] use shortcut connections between single layers that sum up the output feature maps of both layers. This makes filter pruning more complicated since shortcut connections can reactive already pruned channels. Several solutions have been proposed for this problem: In [22, 65], layers with shortcut connections were not pruned to avoid the problem of reactivated channels. Furthermore, in [72, 76], feature maps were sampled in front of each residual block to reduce their dimension. Yet, sampling layers bring additional computation costs. The authors of [23] proposed a group pruning method in which layers connected by a shortcut connection share the same pruning patterns.

In our case, the application of shortcut connections is not a problem as long as the counting functions from Equation 5.5 and Equation 5.7 are implemented correctly: when calculating the layer-wise pruning rates, it must be taken into account whether a shortcut connection is added and if so, whether the inactive channels match on both sides of the shortcut connection. If the network uses shortcut connections in sequential order, such as ResNet-20, it is necessary to verify that the pruned channels match for all sequential shortcut connections. However, since shortcut connections can only reactivate the input channels of layer blocks, their impact on the pruning budget is rather small [22, 65].

5.3 Implementation Details

Algorithm 5.3 summarizes our Holistic Filter Pruning approach to reduce the number of parameters and multiplication of a deep neural network based on a given target size. The pretrained model f_Θ, the target size $\{P^*, M^*\}$, the number of training epochs E, the training data $D = \{(x_i, y_i)\}_{i=1}^{d}$, the batch-size S, the learning loss $\mathcal{L}_{\text{learn}}$, the learning-rate domain $[\eta_0, \eta_E]$, as well as the final value of the regularization parameter λ_E are required as input values. Regarding the learning rate, we use $[\eta_0, \eta_E] = [0.01, 0.0001]$ for the CIFAR-10 classification task, and $[\eta_0, \eta_E] = [0.1, 0.0001]$ for the ImageNet classification.

Sparsity learning (lines 2 to 14): Before each epoch, both the learning rate and the regularization parameter are scheduled. Here, the learning rate is linearly decreased from η_0 to η_E according to the current training epoch (see line 3), and the regularization parameter is linearly increased to λ_E (see line 4). Furthermore, the training data is shuffled and divided into N batches of size S. Training with Holistic Filter Pruning consists of the following steps. First, the current mini-batch is propagated through the model to compute the network prediction (see line 8), which is used to calculate the learning loss (see line 9). Next, the reduction loss is calculated according to the equations 5.4 to 5.7 and added to the learning loss (see lines 10 and 11). The resulting training loss is then derived with respect to the parameters before each parameter is updated in the direction of its negative

Algorithm 3 Holistic Filter Pruning: Reducing the complexity of a deep neural network based on a given target size, which is defined by the number of parameters P^* and multiplication M^* that are available on the target device. The implementation effort is comparatively low since Holistic filter pruning can be integrated seamlessly into the common training procedure of deep neural networks.

Input: Pretrained model f_Θ with $\Theta = \{w_1, \cdot, w_M\}$, Number of Epochs E, Training Data $D = \{(x_i, y_i)\}_{i=1}^{d}$, Batch size S, Learning loss $\mathcal{L}_{\text{learn}}$, Learning-rate domain $[\eta_0, \eta_E]$, Regularization parameter λ_E, Target sizes P^* and M^*.
for $e = 1$ to E **do**
  $\eta \leftarrow \eta_0 - (\eta_0 - \eta_E)\, e/E$
  $\lambda \leftarrow \lambda_E\, e/E$
  Randomly shuffle D.
  Divide D into N batches $\{X_n, Y_n\}_{n=1}^{N}$ of size S.
  for $n = 1$ to N **do**
    $\widehat{Y}_n \leftarrow f_\Theta(X_n)$
    Compute $\mathcal{L}_{\text{learn}}\left(\widehat{Y}_n, Y_n\right)$
    Compute $\mathcal{L}_{\text{reduce}}$ according to Equation 5.4
    Compute $\mathcal{L}_{\text{train}} \leftarrow \mathcal{L}_{\text{learn}} + \lambda\, \mathcal{L}_{\text{reduce}}$
    Compute $\dfrac{\partial \mathcal{L}_{\text{train}}}{\partial w}$
    $w \leftarrow \text{SGD}\left(w, \eta, \frac{\partial \mathcal{L}_{\text{train}}}{\partial w}\right)$
  end for
end for
$f_\Theta \leftarrow \text{Prune}(f_\Theta)$
for $e = 1$ to 3 **do**
  $\eta_e \leftarrow \eta_0 - (\eta_0 - \eta_E)\, e/3$
  Randomly shuffle D.
  Divide D into N batches $\{X_n, Y_n\}_{n=1}^{N}$ of size S.
  for $n = 1$ to N **do**
    $\widehat{Y}_n \leftarrow f_\Theta(X_n)$
    Compute $\mathcal{L}_{\text{learn}}\left(\widehat{Y}_n, Y_n\right)$
    $w \leftarrow \text{SGD}\left(w, \eta_e, \frac{\partial \mathcal{L}_{\text{learn}}}{\partial w}\right)$
  end for
end for
return f_Θ

gradient (see lines 12 and 13). Here, we use SGD optimization with the Nesterov momentum set to 0.9. The training procedure is repeated until the number of epochs is reached.

Pruning and retraining (line 15 to 24): After training using the reduction loss, the channels whose scaling factors are set to zero by the indicator function are completely deleted from the network architecture (see line 16). Subsequently, the remaining channels are retrained for three epochs in order to update the batch statistics of the batch-normalization layers (see lines 17 to 25). Here we reduce the learning rate again from η_0 to η_E.

As noticeable in Algorithm 5.3, the implementation effort of Holistic Filter Pruning is low. Compared to related work, there is no need for solving or approximating complicated optimization problems. On the contrary, the reduction loss can be easily integrated into the common training procedure and the indicator function can be applied to the already existing batch-normalization layers.

5.4 Experiments and Results

In this section, we evaluate our Holistic Filter Pruning (HFP) approach on the classification tasks CIFAR-10 and ImageNet. First, we compare with recent filter pruning methods and show state-of-the-art performance before giving insights into the training procedure. The baselines of experiments on CIFAR-10 are calculated by training for 200 epochs using SGD optimization with the Nesterov momentum set to 0.9 and a batch size of 64. The learning rate is reduced linearly during the training from 10^{-2} to 10^{-4}. For ImageNet, the baselines are taken from the torchvision model zoo*. All experiments are done using Algorithm 5.3. The data sets and architectures used are described in detail in Section 3.

5.4.1 CIFAR-10 with VGG7 and ResNet-56

On the one hand, Table 5.1 shows the pruning results with VGG7 using CIFAR-10. In our first experiment, we specify to prune the number of parameters by 90% and the number of multiplications by 80%. Thus, we achieve comparable pruning rates to HRank [24] but improve the accuracy by almost 3%. In comparison to Zhao *et al.* [79] and SSS [78], we achieve higher pruning rates, while simultaneously increasing the accuracy by more than 1%. Compared to the baseline accuracy, our approach can reduce the number of parameters by 90% with an accuracy drop of only 0.6%. In our second experiment, we can reduce the number of parameters by 95% with only about 1% loss in accuracy.

On the other hand, Table 5.2 shows the pruning results with ResNet-56 using CIFAR-10. We use two different settings with a parameter reduction rate of 50% and 70%, respectively. Thus, our approach is able to prune both the parameters and multiplications by at least

* https://pytorch.org/docs/stable/torchvision/models.html

VGG7 on CIFAR-10

Table 5.1: Top-1 accuracies and percentage reduction in the number of multiplications and parameters for VGG7 (CIFAR-10). Here, a parameter reduction rate of 90% means that 10% of the parameters remain in the reduced model (i.e., the higher the reduction rate the better). Results marked with '-' are not reported by the authors or correspond to the baseline accuracy.

Method	Reduced multiplications [%]	Reduced parameters [%]	Top-1 %
Baseline	-	-	94.89
SSS [78]	41.6	73.8	93.02
Zhao *et al.* [79]	39.1	73.3	93.18
GAL-0.1 [80]	45.2	82.2	90.73
HRank [24]	65.3	82.1	92.34
HRank [24]	76.5	92.0	91.23
HFP	82.0	90.0	**94.21**
HFP	88.0	95.0	**93.86**

ResNet-56 on CIFAR-10

Table 5.2: Top-1 accuracies and percentage reduction in the number of multiplications and parameters for ResNet-56 (CIFAR-10). Here, a parameter reduction rate of 90% means that 10% of the parameters remain in the reduced model (i.e., the higher the reduction rate the better). Results marked with '-' are not reported by the authors or correspond to the baseline accuracy.

Method	Reduced multiplications [%]	Reduced parameters [%]	Top-1 %
Baseline	-	-	93.30
NISP [81]	35.50	42.40	93.01
DCP [69]	47.10	70.30	93.79
GBN-40 [23]	60.10	53.50	93.41
GBN-60 [23]	70.30	66.70	93.07
HRank [24]	50.00	42.40	93.17
HRank [24]	74.10	68.10	90.72
HFP	56.00	50.00	**93.30**
HFP	76.09	71.58	**92.31**

ResNet-50 on ImageNet

Table 5.3: Top-1 accuracies and percentage reduction in the number of multiplications and parameters for ResNet-50 (ImageNet). Here, a parameter reduction rate of 90% means that 10% of the parameters remain in the reduced model (i.e., the higher the reduction rate the better). Results marked with '-' are not reported by the authors or correspond to the baseline accuracy.

Method	Reduced multiplications [%]	Reduced parameters [%]	Top-1 %
Baseline	-	-	76.15
NIPS 2018, NIPS 2019, CVPR 2019			
DCP [69]	55.76	51.45	74.95
FPGM [70]	53.50	-	74.83
GBN-60 [23]	40.54	31.83	76.19
GBN-50 [23]	55.06	53.40	75.18
CVPR 2020			
Hinge [82]	53.45	-	74.70
He *et al.* [83]	60.80	-	74.56
DMCP [84]	73.17	-	74.40
HRank [24]	62.10	-	71.98
HRank [24]	76.04	-	69.10
HFP	55.25	51.01	**76.08**
HFP	70.02	54.93	**75.24**
HFP	73.45	59.20	**74.81**
HFP	78.24	68.48	**74.14**

50% with no loss in accuracy. In comparison to HRank, our HFP approach achieves higher pruning rates with a slightly improved Top-1 accuracy. GBN [23] achieves a slightly higher Top-1 accuracy for comparable pruning rates. However, in our second experiment, we can reduce the number of multiplication by more than 75% with only 1% loss in accuracy, which is the highest reduction rate in comparison.

5.4.2 ImageNet with ResNet-18 and ResNet-50

Table 5.3 shows the pruning results of ResNet-50. To enable accurate comparisons, we evaluate four configurations with various pruning rates and compare with the latest results from CVPR and NIPS. The first configuration of HFP reduces the number of required multiplications by 55% and the number of parameters by 51% with no significant loss in accuracy. The second configuration reduces the number of required multiplications

ResNet-18 on ImageNet

Table 5.4: Top-1 accuracies and percentage reduction in the number of multiplications and parameters for ResNet-18 (ImageNet). Here, a parameter reduction rate of 90% means that 10% of the parameters remain in the reduced model (i.e., the higher the reduction rate the better). Results marked with '-' are not reported by the authors or correspond to the baseline accuracy.

Method	Reduced multiplications [%]	Reduced parameters [%]	Top-1 [%]
Baseline	-	-	69.75
SFP [68]	41.80	-	67.10
FPGM [70]	41.80	-	68.41
HFP	36.30	22.07	**69.15**
HFP	45.00	37.27	**68.53**

by 70% and thus achieves both higher pruning rates and a higher Top-1 accuracy than reported in [23, 24, 82, 83]. Compared to [84], our third configuration slightly improves both the number of multiplications and the Top-1 accuracy. Furthermore, HFP can reduce the number of multiplications of ResNet-50 by 78% and the number of parameters by 68% with only 2% loss in accuracy.

Table 5.4 shows the pruning results of ResNet-18, which is much smaller than ResNet-50, less over-parameterized and consequently more difficult to reduce. Our HFP approach provides new state-of-the-art performance with 36% reduced multiplications and 22% reduced parameters with only 0.6% accuracy decrease. Furthermore, HFP is able to reduce the number of multiplications by 45% with only 1.2% loss in accuracy, which outperforms FPGM [70] and SFP [68] in terms of both accuracy and reduction rates.

5.4.3 Ablation Study on the Regularization Parameter

Table 5.5 shows the pruning results of ResNet-50 intending to prune 60% of the required multiplications and 40% of the parameters by using different values of the regularization parameter λ. As can be seen in Equation 5.9, the regularization parameter controls the weighting between the learning loss on the one hand an the reduction loss on the other: the larger λ, the higher the contribution of the reduction loss whose minimization reduces the model capacity.

The first experiment uses the constant value $\lambda = 1$. As noticeable, the desired pruning rates are not fulfilled after training since the weighting of the reduction loss is to low. The second experiment uses $\lambda = 7.25$, which regularizes the weighting between the learning loss and the reduction loss so that both components have approximately the same magnitude on an untrained model. Here, the desired pruning rates are fulfilled after training.

Ablation study: ResNet-50 on ImageNet

Table 5.5: Ablation study on the effect of the regularization parameter: Top-1 accuracies and percentage reduction in the number of multiplications and parameters for different values of λ. The first two experiments use the constant values 1 and 7.25. During the third experiment, λ is linearly increased from 0 to 7.25.

λ	Reduced multiplications [%]	Reduced parameters [%]	Top-1 %
1.	48	36	**76.41**
7.25	62	42	**75.73**
$0 \rightarrow 7.25$	60	41	**76.08**

However, the accuracy drops below 76% since the imbalance between both losses is high at the beginning of the training since a pretrained model is used for initialization. The third experiment utilizes the proposed strategy of heating up λ from 0 to 7.25 over the training time: The pruning rates are still fulfilled and the accuracy increases in comparison to the second experiment.

5.4.4 Ablation Study on the Pruning Rate Allocation of VGG7

Distributing the resources available to the individual network layers is a well-known problem in filter pruning [23, 45, 75]. In HFP, the budget on parameters and multiplications is distributed automatically across the network depth such that the reduction loss is minimized and the target size is fulfilled. To verify the plausibility of the solutions found, we analyze two different experiments using VGG7 on CIFAR-10: (a) with the objective of pruning 90% of the parameters, and (b) with the objective of pruning 90% of the multiplications. Depending on the objective, the same layers should be pruned to different extents, depending on whether they have comparatively many parameters or comparatively many multiplications. The results of both experiments are visualized in Figure 5.3. On the left y-axis, the pruning rates of the individual layers are shown. Here, a pruning rate of 0.9 means that 90% of the parameters (or multiplications, respectively) of the corresponding layer are pruned. On the right y-axis, the proportional layer sizes in terms of the number of parameters and multiplications are shown. Here, a proportional layer size of 0.9 means that 90% of the total number of parameters (or multiplications, respectively) are located in the respective layer.

VGG7 consists of six convolution layers and two fully-connected layers, see Section 3. As can be seen in Figure 5.3 on the right y-axis, the convolutional layers are particularly expensive in terms of the number of multiplication, while the first fully-connected layer contains the most parameters. In experiment (a), HFP primarily reduces the layers that contribute most to the number of parameters (conv6 and fc7). Especially fc7 has a

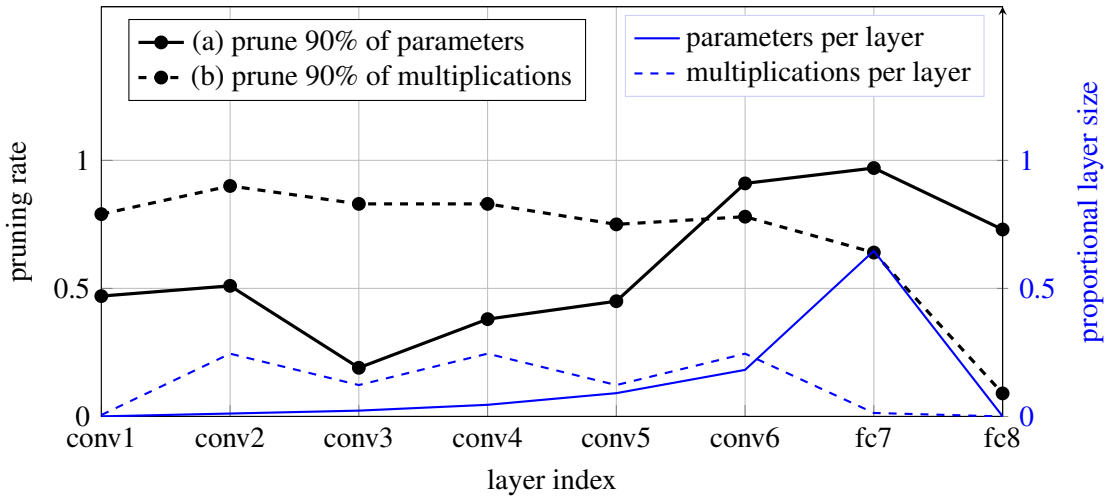

Figure 5.3: Layer-wise pruning rates of VGG7 on CIFAR-10 for two different experiments: (a) with the aim of pruning 90% of the parameters, and (b) with the aim of pruning 90% of the multiplications. On the left-hand side, the pruning rates of the individual layers are shown. Here, a pruning rate of 0.5 means that 50% of the channels (i.e. 50% of the filters or neurons, respectively) are pruned. On the right-hand side, the proportional layers sizes are shown. Here, a proportional layer size of 0.5 means that 50% of the network parameters (or multiplications) are located in the respective layer. Depending on the target reduction, the pruning budget is distributed differently across the individual layers: (a) reduces the layers with many parameters, while (b) especially prunes the convolution layers that have the most multiplications.

large number of parameters and is therefore pruned by approximately 97%. In contrast, experiment (b) mainly leads to a reduction of the convolution layers as they offer more potential for saving multiplications. Consequently, we can first state that HFP distributes flexible pruning rates over the individual layers. Furthermore, the distribution of the pruning budget varies depending on the target reduction. Comparisons with the layer sizes regarding the number of parameters and multiplications result in a meaningful distribution.

5.4.5 Visualization of ResNet-56 on CIFAR-10

This section analyzes how the overall reduction rates of the multiplications and parameters are proportionally distributed among the individual layers for ResNet-56 on CIFAR-10. The final reduction rate is 56% for the number of required multiplications and 50% for the number of parameters. The corresponding Top-1 accuracy is shown in Table 5.2.

Figure 5.4 shows the proportional pruning rates for the multiplications after 1 epoch, 10 epochs, and 100 epochs of training. Here, the proportional pruning rates indicate the contributions of single layers to the overall pruning rate at the current time step. For example, after 10 epochs, 47% of the multiplications are pruned from the model. Of these, 2.3% are proportionally allotted to the first layer. Additionally, the diagram shows

the total number of multiplications of the original layers (dotted line). The three basic blocks (sometimes also called the basic layers) of the ResNet architecture are visible and marked with A, B, and C. Here, one can observe that the proportional pruning rates of the individual layers change over the training time. While in block A the pruning rates decrease as training progresses, the rates in block C increase: the allocation of the pruning budget changes continuously during training. At the end of the training, the second and third block achieve slightly higher pruning rates compared to the first block. However, the differences are comparably small as all intermediate layers share the same number of multiplications.

Analogically, Figure 5.5 shows the proportional pruning rates for the number of parameters. In comparison to the pruned multiplications, the proportional pruning rates of the parameters change significantly depending on the layer index: with an increasing layer size, the pruning rates increase as well. At the end of the training, block C shows the highest contribution to the overall reduction rate of 55%. Thus, the same observation can be made as for the pruning rate allocation of VGG7 (see Section 5.4.4): the layer-wise pruning rates are proportional to the effect that each layer has on reduction loss.

Furthermore, Figure 5.6 shows the Top-1 accuracies of various experiments with different pruning rates using ResNet-56. The performance values are illustrated by colored level curves created by fitting a second-order polynomial function. The baseline accuracy of approximately 93.30% can be maintained with reduction rates up to 50%. Furthermore, one can observe that pruning the parameters has a greater impact on the performance than pruning the multiplications.

5.4.6 Visualization of ResNet-50 on ImageNet

This section analyzes how the overall reduction rates of the multiplications and parameters are proportionally distributed among the individual layers for ResNet-50 on ImageNet. The final reduction rate is 55% for the number of required multiplications and 51% for the number of parameters. The corresponding Top-1 accuracy is shown in Table 5.3.

Figure 5.7 shows the proportional pruning rates for the multiplications after 1 epoch, 10 epochs, and 100 epochs of training. Here, the proportional pruning rates indicate the contributions of single layers to the overall pruning rate at the current time step. For example, after 10 epochs, 46% of the multiplications are pruned from the model. Of these, 3.4% are proportionally allotted to the fourth layer. Additionally, the diagram shows the total number of multiplications of the original layers (dotted line). Initially, a correlation between the layer size and the proportional pruning rates can be observed. As in the case of ResNet-56, the proportional pruning rates change over time. Here, the pruning rates of the first layers decrease, while the pruning rates with layer index 11, 23, and 41 increase. These are exactly the layers that contribute most to the overall number of multiplications.

Figure 5.8 shows the proportional pruning rates for the number of parameters. Here, the

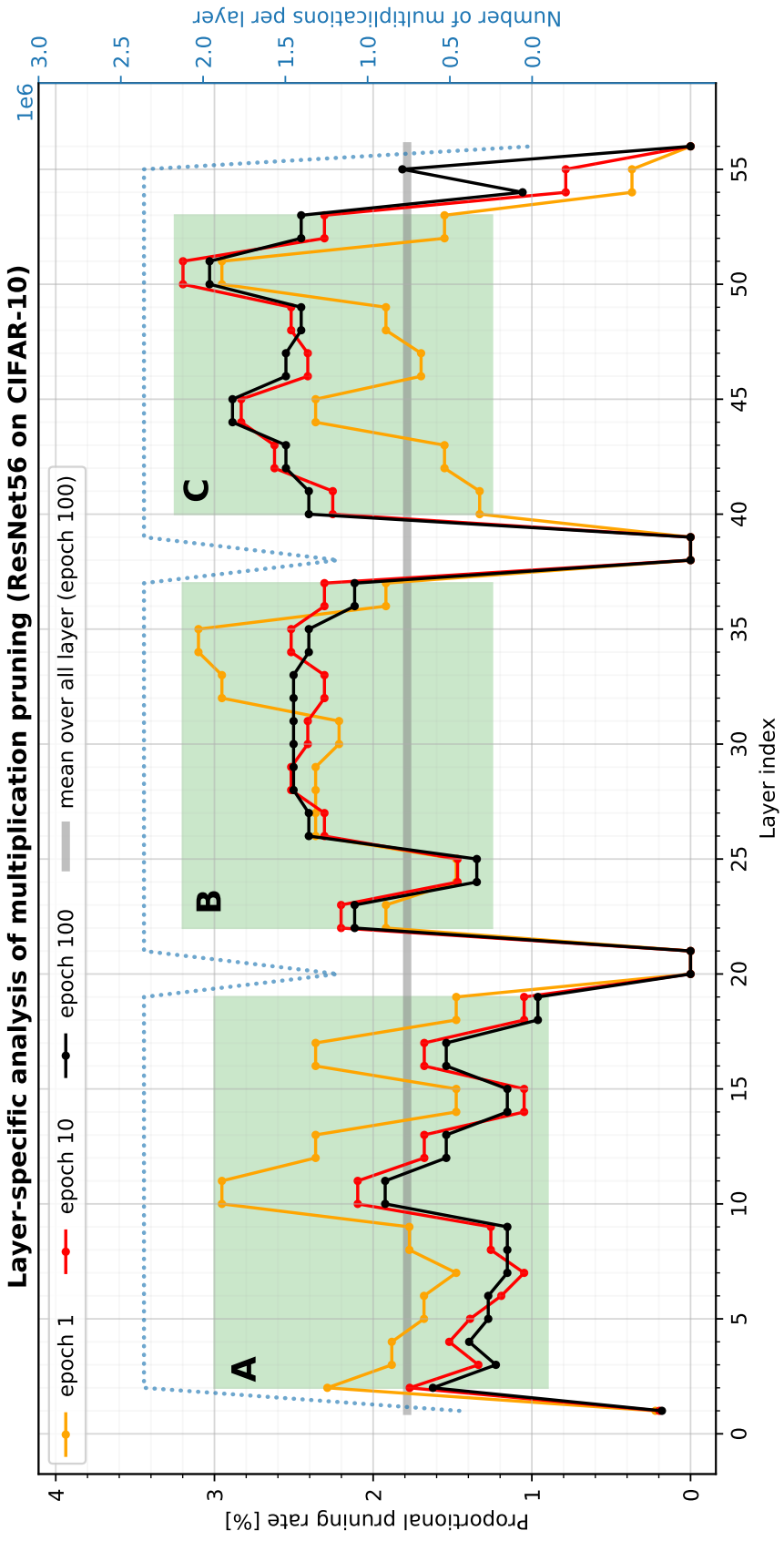

Figure 5.4: Proportional pruning rates for the multiplications after several epochs of training using ResNet-56 on CIFAR-10. The final reduction rates are 56% for the number of multiplications and 50% for the number of parameters. The corresponding Top-1 accuracy is shown in Table 5.2. On the left-hand side, the proportional pruning rates indicate the contribution of single layers to the overall multiplication reduction rate. For example, if 100 multiplications are pruned from the model and the first layer is reduced by 15 multiplications, the proportional pruning rate of the first layer is 15%. On the right-hand side, the original layer sizes are shown. Furthermore, the three basic blocks of the ResNet architecture are marked with A, B, and C.

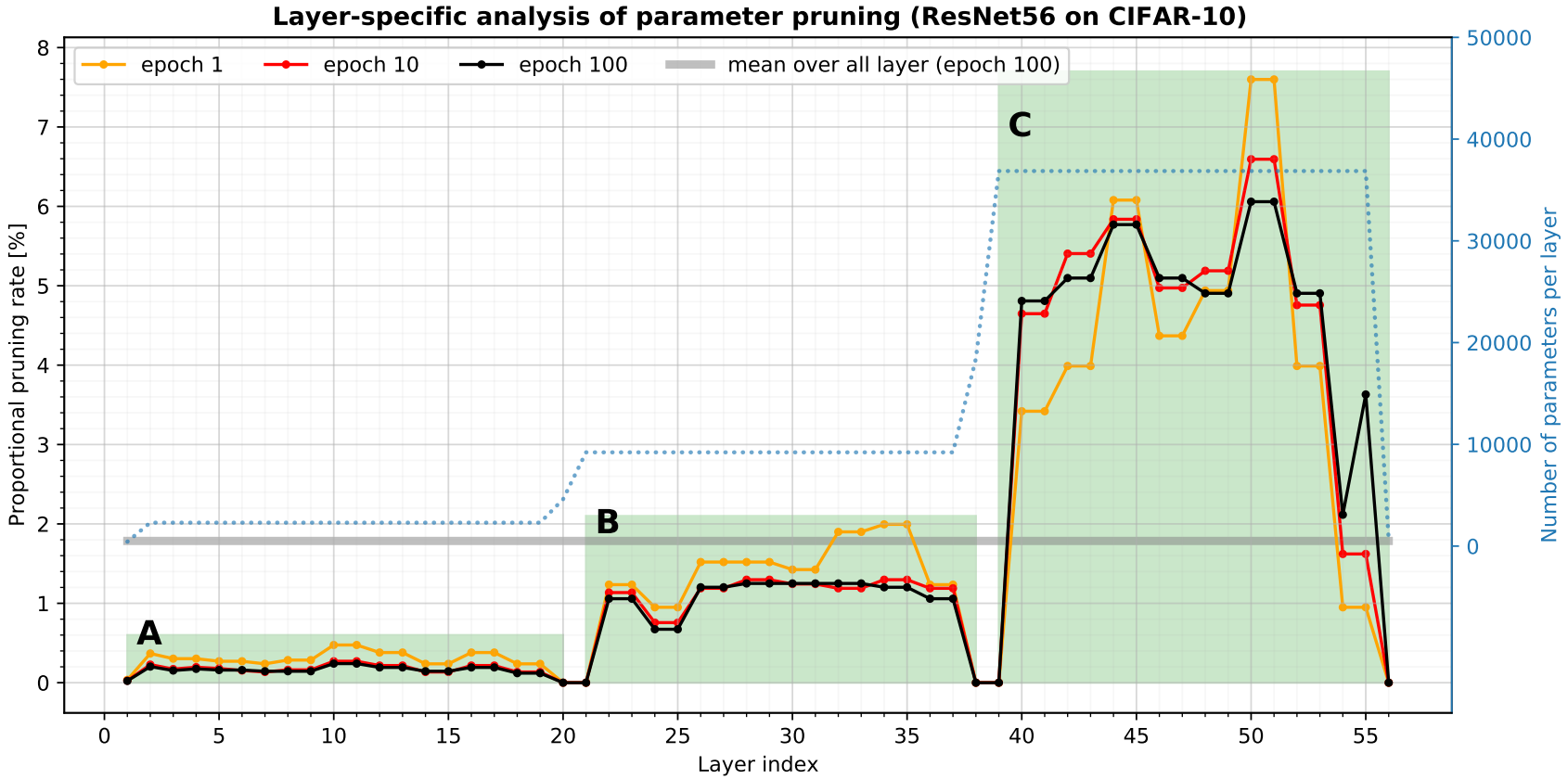

Figure 5.5: Proportional pruning rates for the parameters after several epochs of training using ResNet-56 on CIFAR-10. The final reduction rates are 56% for the number of multiplications and 50% for the number of parameters. The corresponding Top-1 accuracy is shown in Table 5.2. On the left-hand side, the proportional pruning rates indicate the contribution of single layers to the overall parameter reduction rate. For example, if 100 parameters are pruned from the model and the first layer is reduced by 15 parameters, the proportional pruning rate of the first layer is 15%. On the right-hand side, the original layer sizes are shown. Furthermore, the three basic blocks of the ResNet architecture are marked with A, B, and C.

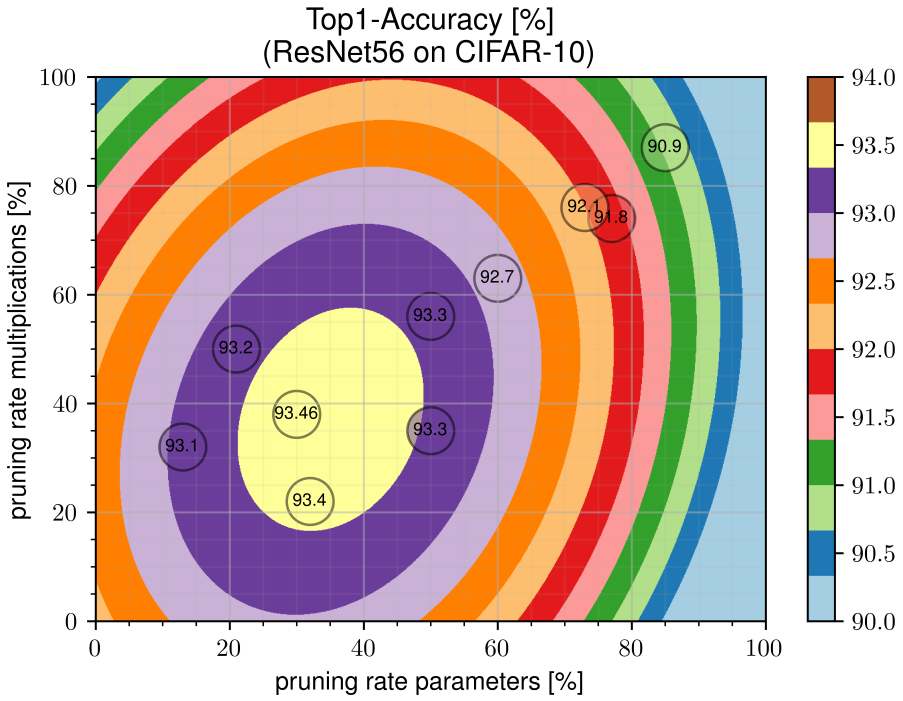

Figure 5.6: Top-1 accuracies of several experiments using ResNet-56 on CIFAR-10. The pruning rates of the parameters and multiplication differ between 20% and 90%. The performance values are illustrated by colored level curves created by fitting a second-order polynomial function.

original layer sizes differ depending on the respective block. Again, the three basic blocks are marked with A, B, and C. With an increasing layer size, the pruning rates increase to the same degree. Since block C has the highest contribution to the total number of parameters, it also shows the highest proportional pruning rates.

In addition, Figure 5.9 shows the Top-1 accuracies of ResNet-50 on ImageNet for different pruning rates. The performance values are illustrated by colored level curves created by fitting a second-order polynomial. HFP can reduce the number of multiplication by up to 60% and the number of parameters by up to 40% with no significant loss in accuracy (76.1% vs 76.15% baseline accuracy). Furthermore, pruning 50% of the multiplications and 28% of the parameters even improves the Top-1 accuracy by approximately 0.5%. This is due to the capability of pruning methods to improve the ability of deep neural networks to generalize. If more than 50% of the parameters are pruned, the Top-1 accuracy drops below 76%. Overall, one can observe that pruning the parameters of ResNet-50 has a greater impact on the performance than pruning the multiplications. Hence, HFP prunes nearly 80% of the multiplication with less than 2% drop in accuracy.

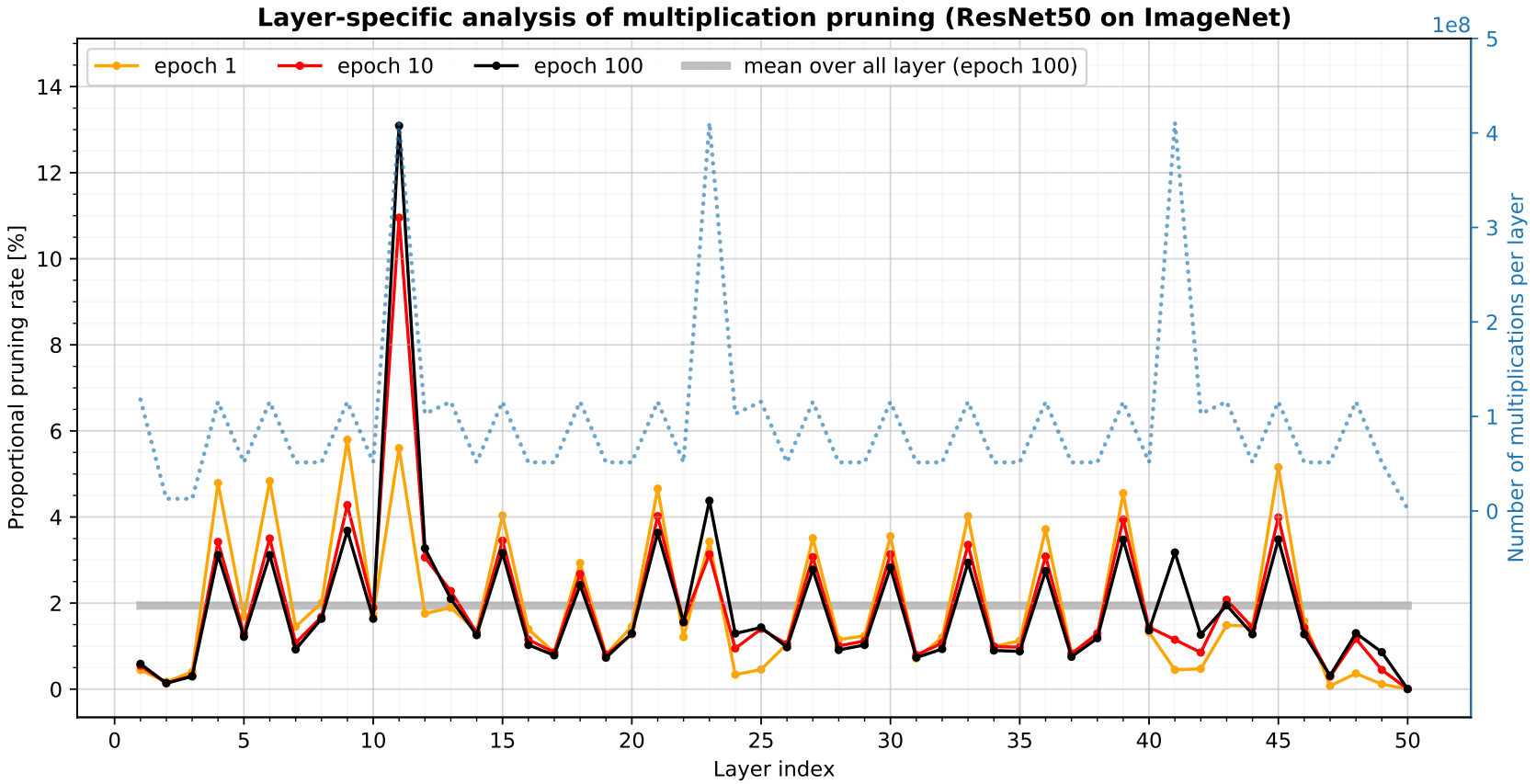

Figure 5.7: Proportional pruning rates for the multiplications after several epochs of training using ResNet-50 on ImageNet. The final reduction rates are 55% for the number of multiplications and 51% for the number of parameters. The corresponding Top-1 accuracy is shown in Table 5.3. On the left-hand side, the proportional pruning rates indicate the contribution of single layers to the overall multiplication reduction rate. For example, if 100 multiplications are pruned from the model and the first layer is reduced by 15 multiplications, the proportional pruning rate of the first layer is 15%. On the right-hand side, the original layer sizes are shown.

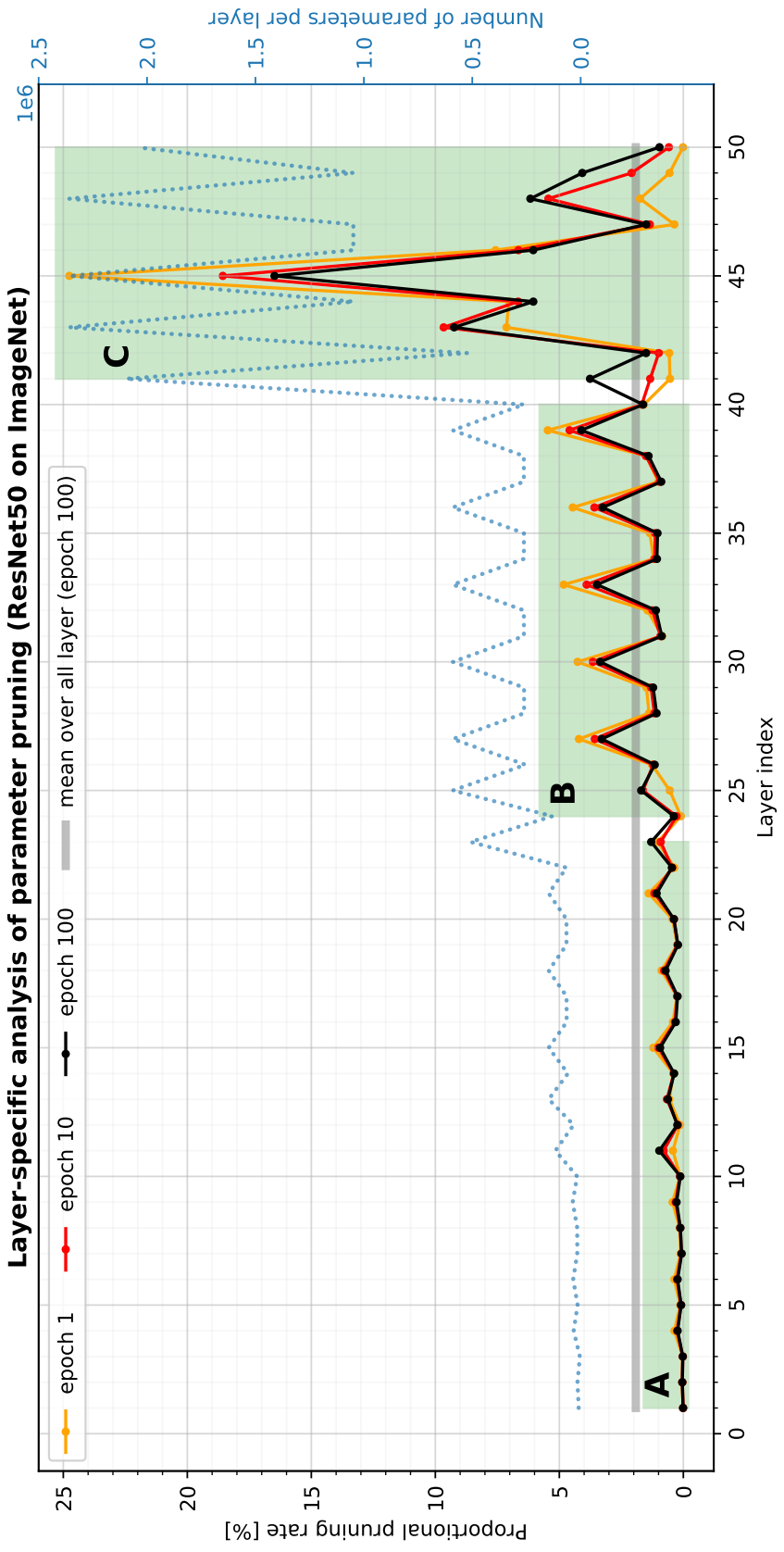

Figure 5.8: Proportional pruning rates for the parameters after several epochs of training using ResNet-50 on ImageNet. The final reduction rates are 55% for the number of multiplications and 51% for the number of parameters. The corresponding Top-1 accuracy is shown in Table 5.3. On the left-hand side, the proportional pruning rates indicate the contribution of single layers to the overall parameter reduction rate. For example, if 100 parameters are pruned from the model and the first layer is reduced by 15 parameters, the proportional pruning rate of the first layer is 15%. On the right-hand side, the original layer sizes are shown. Furthermore, the three basic blocks of the ResNet architecture are marked with A, B, and C.

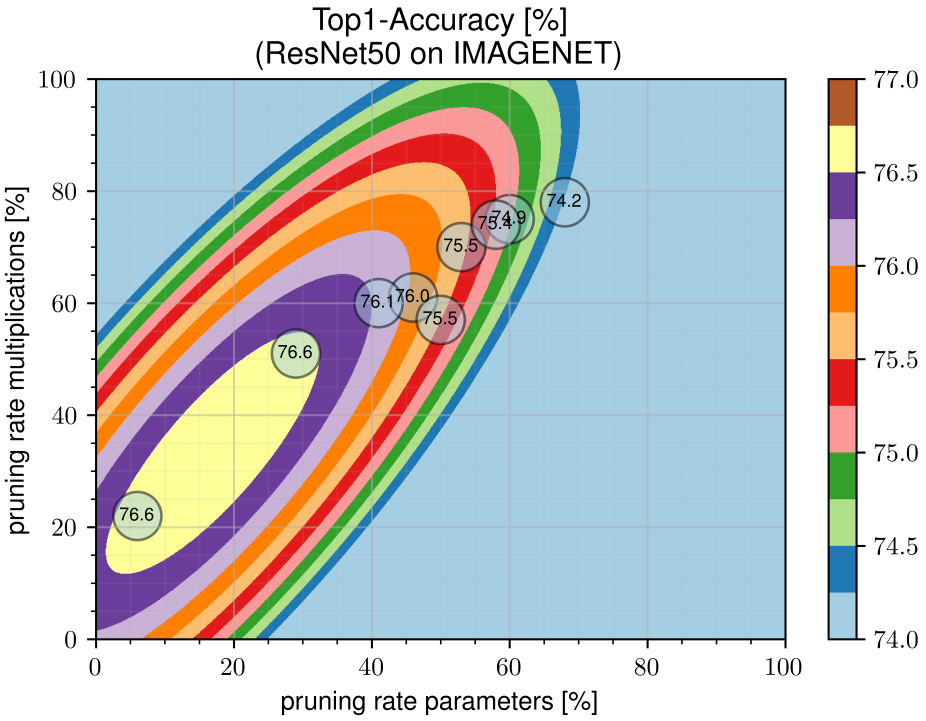

Figure 5.9: Top-1 accuracies of ResNet-50 on ImageNet with different pruning rates. The performance values are illustrated by colored level curves created by fitting a second-order polynomial.

5.5 Related Work

Deep neural networks are usually over-parameterized after the training and have many redundant network connections. These redundancies can be eliminated (e.g. pruned) from the network architecture in order to reduce the model complexity and improve their ability to generalize [10]. The first pruning methods were aimed at setting single weight values to zero in order to trim intermediate layer connections. In this regard, Optimal Brain Damage [85] utilized the second-order derivative of the loss function to calculate saliency scores for each network weight. Subsequently, weights with small saliency scores were pruned iteratively whereas the remaining weights were retrained. Since calculating the second-order derivatives of the loss function with respect to the parameters is too complex for large models, many approaches applied magnitude-based pruning [71, 73, 86, 87].

However, pruning single weights leads to unstructured sparsity, which has no direct benefit on the hardware implementation of deep neural networks since the tensor sizes of both the weights and activations remain unchanged. Thus, the percentage of non-zero weights is an insufficient indicator for rating the complexity of deep neural networks. In contrast, pruning complete filters or neurons from the network architecture leads to structured sparsity, which reduces the tensor sizes of both the weights and activations without the need for specialized hardware [23, 45, 70, 73]. A visualization of the structured

sparsity resulting from filter pruning is given in Figure 5.1. Consequently, filter pruning can also be used to find an appropriate architecture based on a highly over-parameterized model.

According to Figure 1.1, filter pruning can be divided into two subcategories [66]: **saliency based pruning and retraining** on the one hand and **sparsity learning** on the other. Both subcategories are based on pretrained and usually over-parameterized models. These methods are described below. Here, a filter is equivalent to a neuron and a channel.

5.5.1 Saliency-based Pruning and Retraining

Saliency-based pruning methods determine heuristics to calculate saliency scores for each filter of the network. The saliency score indicates the importance of the respective filter: The higher the score the more important the filter is considered to be for fulfilling the learning task. Based on the saliency scores, a certain number or percentage of filters is pruned whereas the remaining ones are retrained. This process is iteratively repeated until the desired network size is reached.

Hu *et al.* identified unimportant filters by analyzing the magnitudes of the output feature maps [67]. Here, feature maps with comparatively small sums of absolute values were considered less important and hence removed. In contrast, Li *et al.* measured the importance of individual filters by calculating the sum over the absolute values of the weights [65]. They argued that filters with small weight values have comparatively less impact on the output of the network and can therefore be pruned. Besides, Zhuang *et al.* argued that distinct channels should have discriminative power: They proposed a ranking heuristic to identify channels with high discriminative power while deleting redundant channels and their corresponding filters [69]. Furthermore, He *et al.* proposed an iterative method in which the weights of filters with a small L^2-norm are first set to zero [68]. However, in the subsequent retraining step, the pruned filters are updated as well to improve the training behavior. The whole procedure is repeated iteratively until the selection of filters with small L^2-norms converges [68]. This selection of filters is finally pruned. Moreover, Yu *et al.* calculated saliency scores by minimizing the reconstruction error in the second-to-last layer before the classification output [81]. Here, Yu *et al.* formulated the resulting optimization problem as a binary integer optimization problem and derived a closed-form solution to prune filters in the front layers as well [81].

However, other saliency-based pruning methods do not use numerical norms to calculate the saliency scores. In [23], Zhonghui *et al.* introduced *Gate Decorator*, a pruning framework that uses gate variables to scale the channel-wise output of intermediate layers. The change in the loss function caused by setting gates variables to zero is calculated using a Taylor expansion. This change is subsequently used for calculating the saliency scores. In [24], Lin *et al.* used the training data to estimate the rank of each feature map of intermediate layers. Subsequently, low-rank feature maps were pruned by deleting the

corresponding filter weights whereas the remaining weights were retrained. Furthermore, He *et al.* proposed a pruning method which is based on the Geometric Median of the filter weights [70]. Here, the Geometric Median was used to identify filters that have the most replaceable contribution regarding the learning task. Thus, He *et al.* pruned the filters that are most replaceable and not the least relevant concerning the learning task.

Recently, Persand *et al.* proposed an approach to combine different strategies for calculating the saliency scores of the filters [88]. They argued that a composition of different saliency scores is more robust than using the same each time [88].

5.5.2 Sparsity Learning

Sparsity learning means integrating sparsity constraints into the training of deep neural networks. Such constraints are aimed at gradually minimizing the influence of individual channels during training. Subsequently, channels with negligible influence can be pruned with no significant loss in accuracy.

Pan *et al.* proposed an approximation of the L^0-norm to penalize incoming and outgoing connections of single filters [89]. After training, filters with small inputs and outputs were pruned. In contrast, He *et al.* proposed a channel selection based on the LASSO regression [72]. Here, pruning individual layers was achieved by minimizing the reconstruction error of the output feature maps of the pruned model and the original model. Furthermore, Liu *et al.* applied an L^1-norm based regularization on the scaling factors of the batch-normalization layers to minimize the influence of single channels during training [76]. After training, a certain percentile of channels with small scaling factors was pruned. Here, Liu *et al.* used a global threshold across all layers. However, the scaling factors were penalized without considering the respective filter size. Furthermore, channels that have already been pruned could not be reactivated again. Besides, Huang *et al.* proposed a try-and-learn algorithm to train pruning agents that identify superfluous filters [90].

Recently, Xiao *et al.* introduced *Auto Prune*, a framework that uses a set of additional parameters to prune single weights or filters during each forward pass [75]. However, in their implementation, the pruning layers are located in front of the batch-normalization layers, reactivating the pruned channels (unless batch-normalization is disabled). Srinivas *et al.* proposed a similar approach using gate variables but neglected batch-normalization layers as well [74].

In [83], He *et al.* proposed a learnable pruning criteria sampler that generates different pruning criteria for different layers. The sampler is differentiable and can therefore be optimized according to the validation loss of the pruned network that is generated based on the sampled pruning criteria. Furthermore, Guo *et al.* modeled the process of filter pruning as a Markov process to efficiently select the filters to be pruned [84]. Since their model is differentiable, it can be optimized according to the learning loss that is used for training. In [82], Li *et al.* combined filter pruning and matrix decomposition by introducing a unified

formulation including sparsity-inducing matrices.

5.6 Conclusion

In this chapter, we proposed Holistic Filter Pruning, an approach for reducing the complexity of deep neural networks by pruning complete filters and neurons from the network architecture. To the best of our knowledge, we were the first to develop a filter pruning method that allows specifying accurate maximum values for both the number of parameters and multiplications. The user is able to define both maximum values depending on the target device: On the one hand, the number of parameters directly results from the amount of storage the network is able to allocate on the target device. On the other hand, the number of required multiplications indicates the computational effort that is needed to evaluate the model on previously unseen data. During training, a reduction loss calculates the difference between the actual model size and the target size in terms of the number of parameters and required multiplications. The reduction loss is then minimized by pruning filters and neurons via the channel-wise affine transformation of the batch-normalization layers. Thus, we also solved common problems in filter pruning: the resources available are distributed automatically across the individual layers so that a global solution is found, and the implementation effort is low since no additional parameters are needed and the reduction loss fits seamlessly into the training procedure of deep neural networks.

In various experiments on the classification tasks CIFAR-10 and ImageNet, our method achieved state-of-the-art performance. Especially for large pruning rates (> 70%), Holistic Filter Pruning yields excellent accuracy and outperforms recent approaches by up to 3%. Furthermore, we investigated how our approach distributes the available resources for different experiments using different network architectures. Here, we found that our Holistic Filter Pruning approach seeks flexible solutions that fit the given target reduction.

Chapter 6

Joint Pruning & Quantization

Deep neural networks dominate current research in machine learning. Due to massive GPU parallelization, the training time is no longer a bottleneck and large models with many parameters lead common benchmark tables. However, mobile devices have finite resources, strictly limiting the memory and computational cost of deep neural networks. In order to reduce complexity, redundant network connections as well as the bit sizes of weights and activations can be reduced. Here, structured sparsity, as achieved by filter pruning, decreases the tensor sizes of weights and activations and is therefore particularly effective in reducing both the memory computational complexity. Furthermore, fixed-point quantization reduces the bit sizes of weights and activations, accelerating the computations on dedicated hardware.

We propose a holistic approach for reducing the complexity of deep neural networks based on four essential metrics: the memory requirement, the computational cost defined by the number of bit operations required during the forward pass, the bandwidth, and the maximum memory cost of the activations. The user is able to specify maximum values for each of these four metrics based on the target device. Our approach then learns both the width and the bit size of each layer so that the maximum values are not exceeded. After training, the batch-normalization layers are folded into the preceding layers, where all weights are encoded using efficient fixed-point quantization. In various experiments, our approach yields excellent performance and we are able to reduce the number of bit operations of ResNet-18 on ImageNet by 55 with no significant loss in accuracy.

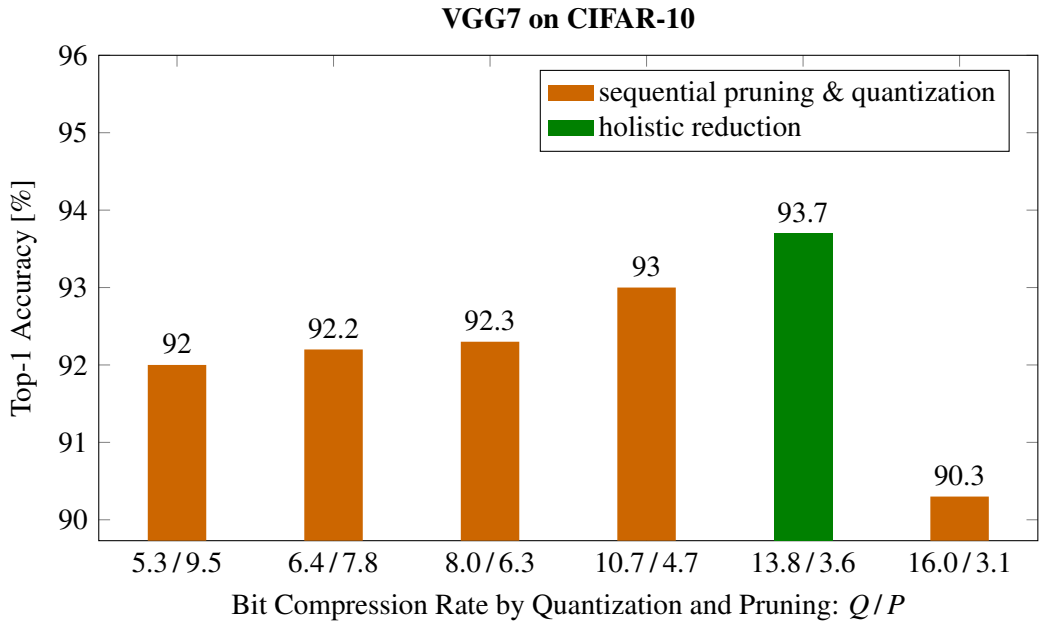

Figure 6.1: Comparison between our holistic reduction approach and different configurations that perform pruning and quantization in sequential order using VGG7 and CIFAR-10. The overall compression rate regarding the number of bits is 50, and all approaches use the same pruning and quantization techniques proposed in Section 6.2. For the sequential approaches, the target compression rate of 50 must be distributed in advance to both pruning and quantization. For example, an overall compression rate of 50 can be achieved by reducing the average bit size of the weights by 8 (via quantization, a compression rate of 8 results from an average bit size of 4 bits) and the number of parameters by 6.3 (via filter pruning). This example belongs to the result of the 3rd column from the left. In contrast, our holistic approach (green bar) automatically learns both the bit size and the number of channels for each layer such that the target compression rate is reached. This saves training time and significantly increases performance.

6.1 Introduction

Deep neural networks are state-of-the-art in many machine learning challenges, outperforming classical methods in computer vision, object detection, and speech recognition [2, 42]. However, a high initial model capacity as well as floating-point operations are necessary to successfully train a deep neural network from scratch [10]. Thus, the greatest results have been accomplished by training large models with many parameters using large-scale data sets [5, 12]. As a result, modern deep neural networks have an extensive memory footprint and consume a lot of energy [12]. Both training and evaluation are thus restricted to powerful processing units.

In contrast, mobile or embedded devices have very limited resources, which must be shared among all mobile applications [12]. Thus, only a fixed number of bits and bit operations are available for individual deep learning applications. This limitation leads

to two major challenges. First, starting with a very limited capacity usually downgrades the training progress of deep neural networks [10]. Instead, many randomly initialized parameters must be provided to successfully train a deep neural network from scratch [10]. Second, distributing the number of available bits and bit operations across the individual layers is a complex and time-consuming problem [91]. Consequently, trained deep neural networks are usually over-parameterized after training and must be significantly reduced when deployed on mobile or embedded devices.

Therefore, reduction techniques have been developed which decrease the complexity of deep neural networks [12]. These approaches can be roughly grouped into two categories: On the one hand, pruning reduces the number of parameters and multiplications by removing redundant network connections. Here, filter pruning directly reduces the tensor sizes of both weights and activations without the need for specialized hardware [22, 91]. On the other hand, quantization reduces the bit sizes of the parameters and activations, whereas fixed-point quantization is particularly efficient on dedicated hardware [13, 18].

However, a combination of filter pruning and fixed-point quantization is essential for finding an efficient deep neural network architecture. Starting with an over-parameterized model, the target size (which is given by a fixed number of bits, e.g. 100 Mbit) can then be achieved by both pruning whole filters and quantizing the remaining. Nevertheless, current approaches that combine pruning and quantization have several drawbacks: [20, 25, 92] only prune single weights, which is ineffective as it leads to unstructured sparsity, [20, 25, 26, 92] omit fixed-point constraints such as bit shifts and integer weights, which are computationally efficient on dedicated hardware, and [20, 25, 26] apply pruning and quantization separately even though both influence each other. For example, there is an infinite number of combinations of pruning and quantization rates that reduce the number of bits of a deep neural network by a factor of 50, see Figure 6.1. Furthermore, [20, 25, 26, 92, 93] use only a single criterion (e.g. the memory requirement), which is used for reduction. This is inappropriate because complexity is always determined by several factors (e.g. by the memory requirement, the bandwidth, and the computational complexity.)

In order to find both an efficient and high-performance architecture starting from an over-parameterized model, we make the following contributions:

- We propose a novel reduction loss consisting of four essential metrics that serve as a measure of complexity: the memory requirement, the computational cost, the bandwidth, and the maximum storage cost of the activations. By setting maximum values for each of those metrics with respect to the target device, the reduction loss becomes an indicator of the difference between the actual and the target complexity.
- We propose a computational graph equipped with custom layers that allows minimizing the reduction loss during training. This is done by changing the architecture using both filter pruning and fixed-point quantization.

- Combined, our contributions result in the ordinary training procedure of reducing a (multi-task) loss by gradient descent optimization. Thus we benefit from all the optimization techniques specially designed for this purpose, such as momentum, running averages, data augmentation, and learning rate schedules.

6.2 Technical Approach: Holistic Reduction

First, we propose a novel reduction loss that calculates the difference between the actual model size and the target size in terms of four essential complexity metrics. Here, the target size results on the one hand from the number of bits and bit operations that are available on the target device, and on the other hand from the bandwidth that is available between the processing unit (e.g. the neural network accelerator) and the corresponding memory unit (e.g. the SRAM or DRAM). Based on the reduction loss, we propose a computational graph equipped with custom layers that allows minimizing the reduction loss during training. Therefore, we propose a novel pruning layer as well as quantization layers that either amplify or replace the original layers. Thus, the network architecture (i.e. the width and bit size of each layer) can be learned such that the target size is fulfilled.

6.2.1 Reduction Loss

Reducing the complexity of deep neural networks first requires a suitable criterion for measuring complexity, which can then be used as an indicator in a reduction method. Neglecting specific constraints on the network architecture, the complexity of a deep neural network is essentially dominated by four criteria, which are described in the following.

Memory requirement: On the one hand, the number of bits required to store a deep neural network depends on both the number of its parameters and their respective bit sizes. Consequently, the memory requirement of a convolutional neural network with L layers can be calculated as follows:

$$\mathrm{Mem} = \sum_{l=1}^{L} C_{l-1}\, C_l\, K_l^2\, B_{w,l}\,. \tag{6.1}$$

Here, l is the layer index, C_l the number of channels in layer l, K_l the kernel size in layer l (if layer l is fully connected, K_l is equal to one), and $B_{w,l}$ the bit size of the weights in layer l. One characterizes this as a layer-wise quantization, which enables a more efficient implementation compared to channel-wise quantization.

Computational complexity: Furthermore, the minimum number of bit operations that are necessary to evaluate a deep neural network in an optimal computer environment can be

calculated by:

$$\text{Bit-Ops} = \sum_{l=1}^{L} C_{l-1}\, C_l\, K_l^2\, H_l\, W_l\, B_{w,l}\, B_{x,l-1}\,. \tag{6.2}$$

Here, H_l and W_l are the height and width of the output feature maps of layer l (if layer l is fully connected, H_l and W_l are both equal to one), and $B_{x,l-1}$ is the bit size of the activation function in layer $l-1$. Consequently, the number of required bit operations results from the number of multiply-and-accumulate operations (MACs) multiplied with the respective bit sizes of the weights and activations. This metric has also been used in [93, 94, 95] and assumes an optimal processing unit to specify a lower bound for the computational complexity during the inference.

Bandwidth: On the other hand, the bandwidth of a deep neural network indicates the amount of data traffic that originates between the processing unit (e.g. the deep neural network accelerator) and the memory unit (e.g. the DRAM or SRAM [12]). Consequently, the bandwidth depends on both the size of the output feature maps and their respective bit sizes:

$$\text{Bandwidth} = \frac{1}{T} \sum_{l=1}^{L} C_l\, H_l\, W_l\, B_{x,l}\,. \tag{6.3}$$

Here, T is the cycle duration that depends on the actual learning task. For example, in an object detection framework, a frame rate of 20fps (frames per second) results in a cycle duration of 0.05s. The cycle duration is usually a fixed quantity, which remains unchanged between the original model and the reduced model.

Maximum storage activations: Furthermore, the maximum memory requirement of the activations can also be a critical factor when evaluating the network on embedded devices [12, 77]. Therefore, we use an additional metric and calculate the maximum storage cost for the activations according to

$$\text{Max-Storage} = \max\left(C_1 H_1 W_1 B_{x,1},\, \ldots,\, C_L H_L W_L B_{x,L}\right). \tag{6.4}$$

Based on these four complexity metrics, we define the reduction loss as follows:

$$\mathcal{L}_{\text{reduce}} = \sum_{i=1}^{4} \text{ReLU}\left(\frac{\text{Criterion}_i - \text{Criterion}_i^*}{\text{Criterion}_i^0}\right), \tag{6.5}$$

$$\text{Criterion}_i \in \{\text{Mem, Bit-Ops, Bandwidth, Max-Storage}\}$$

Here, Criterion_i denotes the corresponding complexity metric from the equations 6.1 to 6.4. Furthermore, Criterion_i^0 describes the respective complexity of the original model, and

Criterion_i^* specifies the target complexity resulting from the target device. For example, Mem^0 denotes the number of bits of the original model, and Mem^* specifiers the number of bits that are available on the target device. Hence, the terms within the rectifier function indicate the normalized differences between the actual mode complexity and the target complexity. The rectifier function is used to limit the reduction loss at zero. In contrast to the mean squared error, the rectifier function has the advantage of only penalizing deviations greater than zero. Since the differences are normalized to the range [0,1], they can be accumulated to calculate the total deviation between the actual model complexity and the target complexity.

In Equation 6.5, the original model complexities Criterion_i^0 as well as the target complexities Criterion_i^* are constant values. In contrast, the actual model complexities Criterion_i can be adjusted during training: quantization reduces the bit sizes of the weights and activations, while filter pruning reduces the number of active channels. In the following, we will design custom layers so that the bit sizes $B_{w,l}$ and $B_{x,l}$ as well as the width C_l of layer l can be learned during the training to minimize the reduction loss.

According to Equation 1.1, the reduction loss can be integrated into the common training procedure of deep neural networks as follows:

$$\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda \, \mathcal{L}_{\text{reduce}} \, . \tag{6.6}$$

Here, $\mathcal{L}_{\text{learn}}$ is the learning loss that represents the learning task, $\mathcal{L}_{\text{train}}$ is the overall training objective, and λ is the regularization parameter that scales the weighting between both losses. Hence, the regularization parameter is chosen so that the learning loss and the scaled reduction loss have approximately the same magnitude. The same procedure for combining different loss functions has also been used in [77, 91]. Therefore, we initialize λ so that $\lambda \, \mathcal{L}_{\text{reduce}}$ is equal to the expectation value of the learning loss over the training set. For example, if the average cross-entropy loss for an untrained model is 3 on the CIFAR-10 classification task, and the reduction loss has an initial value of 2, λ is equal to 1.5.

6.2.2 Pruning Layer

In [91], Enderich *et al.* pruned complete filters and neurons from the network architecture by minimizing the scaling factors of the batch-normalization layers during training. However, this has several drawbacks. First, batch-normalization layers have a fixed location after convolutional or fully-connected layers and therefore have difficulty pruning shortcut connections. Second, filters with small scaling factors must be manually set to zero after the training and the remaining ones must be retrained [91]. Third, some network architectures do not include batch-normalization layers or have to freeze them [97].

Therefore, we propose a novel pruning layer $f(\cdot)$ that can be included at any layer index l of the network architecture. The pruning layer has a trainable scalar $\rho_{l,c}$ for each channel

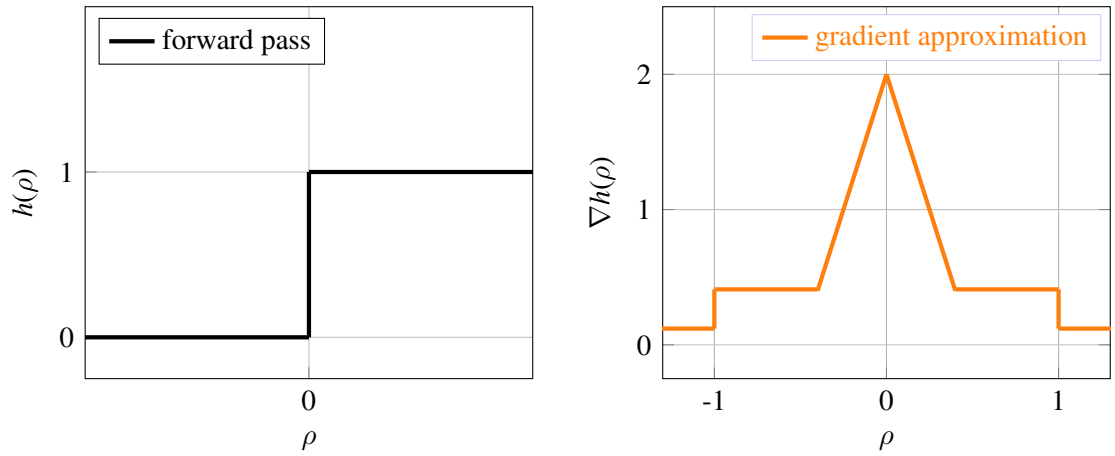

Figure 6.2: During the forward pass, $h(\rho)$ outputs one if ρ is positive and zero otherwise. During the backward pass, a long-tailed estimator is used to approximate the gradient of $h(\rho)$ with respect to ρ. Within the range [–0.4, 0.4], a piece-wise polynomial function approximates the gradient of the non-differentiable step of the Heaviside function [96]. Outside the range [–1, 1], the gradient is significantly reduced to address the saturation effect. However, since ρ is a real-valued parameter with an open range of values, we use a non-vanishing gradient of 0.1 to continue learning.

c in layer l. During each forward pass, the pruning layer multiplies each channel input $x_{l,c}$ with the output of the Heaviside step function $h(\rho_{l,c})$ according to

$$f(x_{l,c}, \rho_{l,c}) = x_{l,c}\, h(\rho_{l,c}) = \begin{cases} 0 & \text{if } \rho_{l,c} \leq 0 \\ x_{l,c} & \text{if } \rho_{l,c} > 0 \end{cases} . \tag{6.7}$$

Thus, the pruning layer deactivates single channels if the corresponding parameter is smaller than or equal to zero.

However, as noticeable in Figure 6.2, the derivative of the Heaviside function is zero almost everywhere, which makes optimization by gradient descent infeasible. Therefore, the gradient of the Heaviside function must be estimated during each backward pass. This has been thoroughly discussed in [96], in which Liu *et al.* recommended using a long-tailed estimator. However, we modify their proposed approximation by using a non-vanishing gradient as follows:

$$\frac{\partial\, h(\rho_{l,c})}{\partial \rho_{l,c}} := \begin{cases} 2 - 4|\rho_{l,c}| & \text{if } |\rho_{l,c}| < 0.4 \\ 0.4 & \text{if } 0.4 \leq |\rho_{l,c}| \leq 1 \\ 0.1 & \text{if } |\rho_{l,c}| > 1 \end{cases} . \tag{6.8}$$

Within the range [–0.4, 0.4], a piece-wise polynomial function approximates the gradient of the non-differentiable step of the Heaviside function [96]. Outside the range [–1, 1],

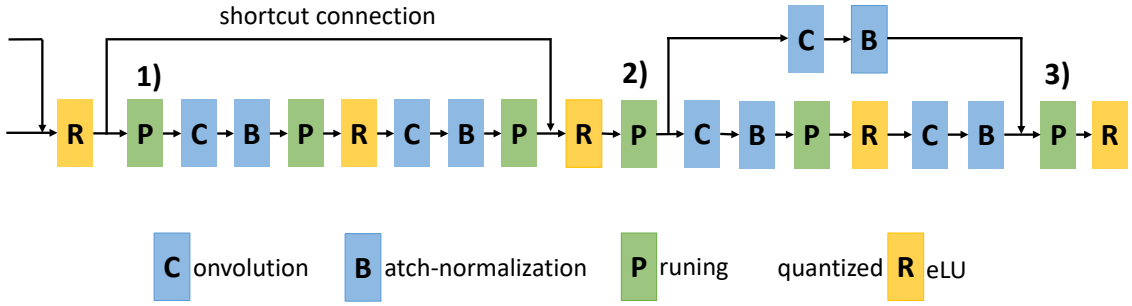

Figure 6.3: Illustration of the computational graph during training. The pruning layers are included after the batch-normalization layers. However, if the network uses shortcut connections, an additional pruning layer is inserted after the shortcut connection to prune the input dimension of the following layer (see layer no. 1). If the shortcut connection itself has a computation layer, the additional pruning layer is inserted in front of the shortcut to prune both input dimensions (see layer no. 2). Furthermore, the last pruning layer of the respective block is shifted behind the shortcut to simultaneously prune both output channels (see layer no. 3). In this way, different channels can be propagated through the shortcut connections, which is also different from the method proposed in [75]. After training, the pruning layers that are located right behind batch-normalization layers can be folded into the preceding convolutional or filly-connected layers.

the gradient is significantly reduced to address the saturation effect. However, since ρ is a real-valued parameter with an open range of values, we use a non-vanishing gradient of 0.1 to continue learning.

Using the Heaviside function of the pruning layer, the number of active channels C_l in layer l (see equations 6.1 to 6.4) can be calculated as follows:

$$C_l = \sum_c h(\rho_{l,c}) = \|h(\rho_l)\|_1 \,. \tag{6.9}$$

Location & Shortcut Connections: In general, the pruning layers are inserted after convolutional or fully-connected layers. However, if batch-normalization is applied, the pruning layers must be inserted after the batch-normalization layers to prune their affine transformations as well.

However, some deep neural network architectures such as ResNet [5] or DenseNet [35] use shortcut connections between single layers to sum up the corresponding output feature maps. A visualization is given in Figure 6.3. In the case of filter pruning, shortcut connections can reactivate already pruned channels and must also be taken into account. Therefore, we place an additional pruning layer after the shortcut connection to prune the input dimension of the following basic block (see layer no. 1 in Figure 6.3). Furthermore, if the shortcut itself has a computational layer, we place the pruning layer in front of the shortcut connection to prune both the input dimension of the basic block and the input dimension of the computational layer of the shortcut connection (see layer no. 2). In addition, the last pruning layer of the respective block is shifted behind the shortcut

connection to simultaneously prune both output dimensions (see layer no. 3).

Initialization: In general, we recommend to initialize the trainable parameters ρ of the pruning layers with 0.5, which does not change the computation of the forward pass in the first place. However, to speed up training, we recommend using information from the corresponding convolutional and batch-normalization layers. The most common and intuitive assumption in filter pruning states that weights with relatively small absolute values are less important for fulfilling the learning task [65, 67, 68]. The same principle can be applied to the scaling factors of the batch-normalization layers [76]. Thus, we calculate the mean absolute value of the weights of each filter and multiply it with the absolute value of the scaling factor of the corresponding batch-normalization layer. For each layer, we normalize the resulting values to the range (0,1], which gives a first indication of the importance of individual filters in the respective layer. Subsequently, we use the normalized values as an initialization for the trainable parameters of the corresponding pruning layer (i.e. the pruning layer that is located right after the batch-normalization layer.)

6.2.3 Fixed-Point Quantization Layers

A quantization function Q maps an input signal x to a smaller set of discrete values $\widetilde{x}$. However, three additional constraints enable to efficiently store $\widetilde{x}$ as fixed-point number: **symmetric** quantization functions do not shift zero-points and therefore avoid cross-terms within matrix-multiplications [18, 44], **power-of-two** step-sizes enable bit-shift operations [13, 18, 44], and **per-tensor** step-sizes share the same value across all entries in a tensor and therefrom enable group-wise bit-shifts [18, 44]. The corresponding fixed-point quantization function can therefore be written as

$$\widetilde{x} = Q(x, f, B) = \text{clip}\left(\left\lfloor \frac{x}{2^{-\lfloor f \rceil}} \right\rceil, \min_B, \max_B\right) 2^{-\lfloor f \rceil} \tag{6.10}$$

$$= \text{clip}\left(x_I, \min_B, \max_B\right) 2^{-\lfloor f \rceil} . \tag{6.11}$$

Here, x_I is rounded to integer values, $\lfloor \cdot \rceil$ rounds to the closest integer, $\lfloor f \rceil$ is the position of the decimal point, $2^{-\lfloor f \rceil}$ is the uniform step-size, and $[\min_B, \max_B]$ is the dynamic range of the quantization function that depends on both the bit size B and the sign:

$$[\min_B, \max_B] = \begin{cases} [0, 2^{\lfloor B \rceil} - 1] & \text{if } \widetilde{x} \text{ is unsigned} \\ [-2^{\lfloor B \rceil - 1}, 2^{\lfloor B \rceil - 1} - 1] & \text{if } \widetilde{x} \text{ is signed} \end{cases} . \tag{6.12}$$

In order to learn an individual quantization for each layer, f and B are trainable parameters with a continuous range of values that must be rounded to integers during each forward pass. Otherwise, optimization with gradient descent would be unfeasible. As in the case of

the Heaviside function, the derivative of the rounding function is zero almost everywhere. Therefore, we utilize the straight-through estimator [53] to approximate its local gradient as follows:

$$\frac{\partial \lfloor x \rceil}{\partial x} := 1 \,. \tag{6.13}$$

Thus, the derivatives of the unsigned quantization function $\left([\min_B, \max_B] = [0, 2^{\lfloor B \rceil} - 1]\right)$ with respect to x, B, and f can be calculated as follows:

$$\frac{\partial \widetilde{x}}{\partial x} = \begin{cases} 1 \text{ if } \min_B < x_I < \max_B \\ 0 \text{ else} \end{cases} , \tag{6.14}$$

$$\frac{\partial \widetilde{x}}{\partial B} = \begin{cases} \log(2)\, 2^{-\lfloor f \rceil}\, 2^{\lfloor B \rceil} \text{ if } x_I > \max_B \\ 0 \text{ else} \end{cases} , \tag{6.15}$$

$$\frac{\partial \widetilde{x}}{\partial f} = \begin{cases} \log(2)\,(x - x_q) \text{ if } \min_B < x_I < \max_B \\ -\log(2)(2^{\lfloor B \rceil} - 1)2^{-\lfloor f \rceil} \text{ if } x_I > \max_B \\ 0 \text{ else} \end{cases} . \tag{6.16}$$

In the case of a signed quantization function $\left([\min_B, \max_B] = [-2^{\lfloor B \rceil - 1}, 2^{\lfloor B \rceil - 1} - 1]\right)$, the derivatives with respect to x, B, and f are:

$$\frac{\partial \widetilde{x}}{\partial x} = \begin{cases} 1 \text{ if } \min_B < x_I < \max_B \\ 0 \text{ else} \end{cases} , \tag{6.17}$$

$$\frac{\partial \widetilde{x}}{\partial B} = \begin{cases} \log(2)\, 2^{-\lfloor f \rceil}\, 2^{\lfloor B \rceil - 1} \text{ if } x_I > \max_B \\ -\log(2)\, 2^{-\lfloor f \rceil}\, 2^{\lfloor B \rceil - 1} \text{ if } x_I < \min_B \\ 0 \text{ else} \end{cases} , \tag{6.18}$$

$$\frac{\partial \widetilde{x}}{\partial f} = \begin{cases} \log(2)\,(x - x_q) \text{ if } \min_B < x_I < \max_B \\ -\log(2)(2^{\lfloor B \rceil - 1} - 1)2^{-\lfloor f \rceil} \text{ if } x_I > \max_B \\ \log(2)(2^{\lfloor B \rceil - 1})2^{-\lfloor f \rceil} \text{ if } x_I < \min_B \end{cases} . \tag{6.19}$$

During each forward pass, the signed fixed-point quantization function quantizes the network parameters whereas the ReLU activation function is replaced by an unsigned

fixed-point quantization function. Therefore, each layer has its quantization parameters. Consequently, the bit sizes of the parameters B_w and activations B_x from the equations 6.1 and 6.4 are calculated by:

$$\lfloor B_{w,l} \rceil \quad \text{and} \quad \lfloor B_{x,l} \rceil \ . \tag{6.20}$$

Initialization: The initialization of the bit sizes B_w and B_x depends on the target reduction: the higher the reduction rate the lower the initial bit sizes. We recommend initializing with 8 bits unless the reduction rate is greater than 80%, where initialization is done with 6 bits. If the reduction rate is greater than 90%, initialization is done with 4 bits. The positions of the decimal points are initialized such that the mean squared quantization error is minimized on pretrained weights.

6.3 Implementation Details

Algorithm 4 summarizes our Holistic Reduction approach to train and simultaneously reduce deep neural networks based on four essential criteria. The pretrained model f_Θ, the target complexities $\{\text{Mem}^*, \text{Bit-Ops}^*, \text{Bandwidth}^*, \text{Max-Storage}^*\}$, the number of training epochs E, the training data $D = \{(x_i, y_i)\}_{i=1}^{d}$, the batch-size S, the learning loss $\mathcal{L}_{\text{learn}}$, as well as the regularization parameter λ are required as input values. Here, we train for 100 epochs on the ImageNet classification task, and for 250 epochs on the CIFAR-10 classification task. The batch size is 128 and the learning loss is Categorical the Cross Entropy from Equation 2.15.

Before the training begins, the pruning layers are inserted according to the explanations in Section 6.2.2. Furthermore, the computations of the convolutional layers, the fully-connected layers, and the ReLU activation functions are provided with the fixed-point quantization functions from Section 6.2.3. Subsequently, the bit sizes are initialized according to the target reduction rate, and the decimal points of the quantization functions are initialized so that the resulting quantization error is minimized based on the pretrained model (see lines 4 to 8). Here, a subset of the training data is used to calculate the layer activations and their quantized counterparts.

Before each epoch, the training data is shuffled and divided into N batches of size S. Before each training step, the learning rate is scheduled using the One-Cycle method proposed in [98] (see line 14). Here, the maximum value for the learning rate is 10^{-3}, the division factor is 25, and the final division factor is 100. Next, the fixed-point quantization function from Equation 6.11 quantizes both the parameters and activations during the first forward pass to update the batch statistics of the batch-normalization layers (see line 15). Subsequently, the batch-normalization layers are folded into the preceding convolutional or fully-connected layers according to Equation 2.9 (see line 16). During the second forward pass, the folded weights as well as the layer activations are quantized using the

Algorithm 4 Holistic Reduction: We train and simultaneously reduce the complexity of a deep neural network based on state-of-the-art optimization techniques. Here, complexity is defined by four essential metrics: the memory requirement, the computational effort, the bandwidth, and the maximum storage of the activations. By setting maximum values for each metric, a reduction loss can be calculated that indicates the difference between the actual and the target complexity.

Input: Pretrained model f_Θ, Number of Epochs E, Training Data $D = \{(x_i, y_i)\}_{i=1}^{d}$, Batch size S, Learning loss $\mathcal{L}_{\text{learn}}$, Learning-rate domain $[\eta_0, \eta_E]$, Regularization parameter λ, Target complexities {Mem*,Bit-Ops*,Bandwidth*,Max-Storage*}.
Insert pruning layers according to Section 6.2.2
Equip quantization layers according to Section 6.2.3
for $l = 1$ to L **do**
    Initialize $B_{w,l}$ and $B_{x,l}$ with 4 bits.
    Initialize $f_{w,l} \in \mathbb{Z}$ and $f_{x,l} \in \mathbb{Z}$ s.t.
    $\left|\left|\widehat{w}_l - Q^S\left(\widehat{w}_l, f_{w,l}, B_{w,l}\right)\right|\right|^2$ and
    $\left|\left|x_l - Q^U\left(x_l, f_{x,l}, B_{x,l}\right)\right|\right|^2$ are minimized.
end for
for $e = 1$ to E **do**
    Randomly shuffle D.
    Divide D into N batches $\{X_n, Y_n\}_{n=1}^{N}$ of size S.
    for $n = 1$ to N **do**
        $\eta \leftarrow$ OneCycleLR
        Forward pass: Update batch statistics
        Fold the BN layers according to Equation 2.9
        Forward pass: $\widehat{Y}_n \leftarrow f_\Theta(X_n)$
        Compute $\mathcal{L}_{\text{learn}}(\widehat{Y}_n, Y_n)$
        Compute $\mathcal{L}_{\text{reduce}}$ according to Equation 6.5
        Compute $\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{learn}} + \lambda\, \mathcal{L}_{\text{reduce}}$
        Compute $\frac{\partial \mathcal{L}_{\text{train}}}{\partial w}$
        $w \leftarrow \text{Adam}\left(w, \eta, \frac{\partial \mathcal{L}_{\text{train}}}{\partial w}\right)$
    end for
end for
Fold the BN layers according to Equation 2.9
Quantize the folded weights according to Equation 6.11
return reduced model

fixed-point quantization function from Equation 6.11 (see line 17). Since the last layer is usually not followed by a batch-normalization layer, the weights of the last layer are used for quantization. After calculating both the learning loss and the reduction loss, the overall training objective is derived with respect to the model parameters by using the gradient estimators from Equation 6.8 and Equation 6.13 (see line 21). Subsequently, each parameter is updated in the direction of its negative gradient (see line 22). Here, we use the Adam optimizer with the beta coefficients set to (0.9, 0.999) [27, 37]. The training procedure is repeated until the number of epochs is reached.

After training, the batch-normalization layers are folded into the preceding convolutional or fully-connected layers, and the weights of the folded layers are quantized.

6.4 Experiments and Results

In this section, we evaluate our Holistic Reduction approach on the benchmark data sets CIFAR-10 and ImageNet. First, we compare with state-of-the-art reduction techniques that are capable of reducing at least one of the four complexity metrics defined in the equations 6.1 to 6.4. However, most of the related approaches focus only on single complexity metrics (e.g. on the number of required bit operations), which is why we conduct several experiments to provide adequate comparisons. Second, we provide an ablation study on the benefits of our holistic reduction approach compared to sequential pruning and quantization. Furthermore, we give insights into the training procedure and analyze the allocation of both the bit sizes and the layer widths. All experiments are done using Algorithm 4. The data sets and architectures used are described in detail in Section 3.
Note for all figures and tables: For example, a compression rate of 50 corresponds to a reduction of 98% (which means that only 2% of the respective quantity remains in the reduced model).

6.4.1 Reducing the Number of Bit Operations

In this section, we reduce the number of bit operations, which is calculated according to Equation 6.2. On the one hand, Figure 6.4 (a) shows the Top-1 accuracies as well as the compression rates of VGG7 on CIFAR-10. Here, our approach can reduce the number of bit operations by a factor of 215 with only about 1% loss in accuracy (93.25% vs 94.39% baseline accuracy). Compared to DJPQ [93], which is a recently published joint pruning and quantization approach (ECCV 2020), we improve the Top-1 accuracy by almost 2% (93.25% vs. 91.43%) at the same compression rate of 215. Compared to RQ [95] and WAGE [99], we achieve a comparable Top-1 accuracy (93.25% vs. 93.22% and 92.04%, respectively) while increasing the compression rate from 64 to 215. Thus, our approach yields either a higher reduction rate at a comparable accuracy or a higher accuracy at a

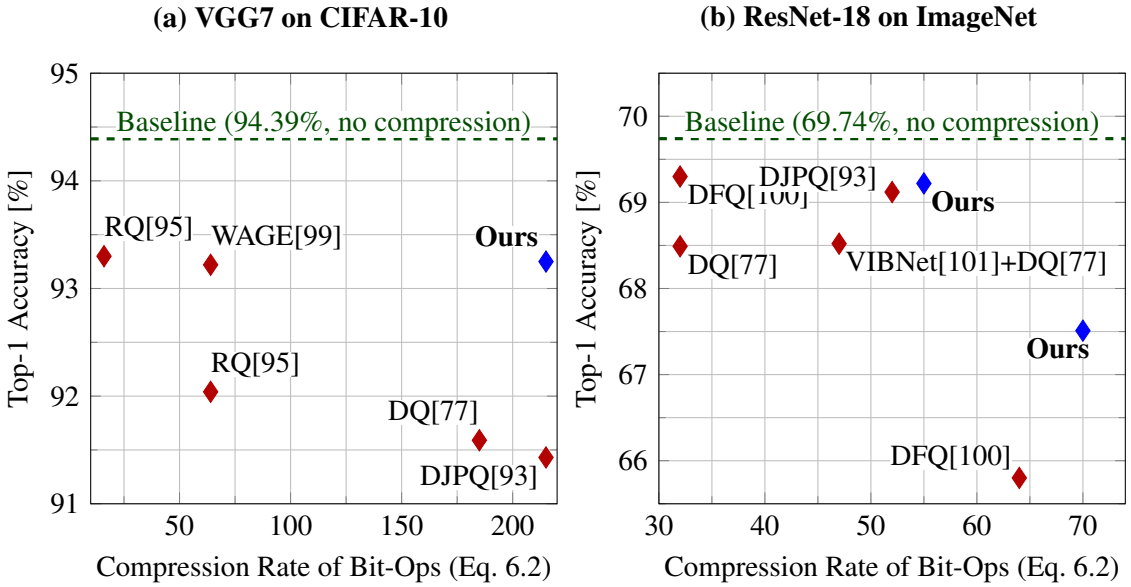

Figure 6.4: Accuracies and compression rates of different reduction methods that reduce the number of **bit operations** (see Eq. 6.2) of VGG7 and ResNet-18 on CIFAR-10 and ImageNet.

comparable reduction rate.

On the other hand, Figure 6.4 (b) shows the Top-1 accuracies as well as the compression rates of ResNet-18 on ImageNet. Here, our approach reduces the number of bit operations by a factor of 55 with only 0.5% loss in accuracy (69.22% vs. 69.74% baseline accuracy). Furthermore, we achieve a compression rate of 70 with only about 2.2% loss in accuracy. Compared to DJPQ [93], we slightly improve both the Top-1 accuracy (69.22% vs. 69.12%) and the compression rate (55 vs. 52).

6.4.2 Reducing the Number of Bits

In this section, we reduce the number of bits, which is calculated according to Equation 6.1. On the one hand, Figure 6.5 (a) shows the Top-1 accuracies as well as the compression rates of ResNet-56 on CIFAR-10. Our approach reduces the number of bit operations by a factor of 6.6 with no significant loss in accuracy (93.10% vs. 93.30% baseline accuracy). Compared to GBN [23], which achieves a comparable Top-1 accuracy (93.10% vs. 93.07%), we significantly improve the compression rate from 3 to 6.6. DCP [69] is the only method that achieves a higher accuracy of 93.79%. However, DCP also uses a higher baseline of 93.8% for initialization.

On the other hand, Figure 6.5 (b) shows the Top-1 accuracies as well as the compression rates of ResNet-18 on ImageNet. Here, we compare with Guerra *et al.* [26], who recently proposed an approach for sequential pruning and quantization. In their experiments, they first apply a quantization method such as BinaryConnect [46] or DoReFa-Net [14] and then

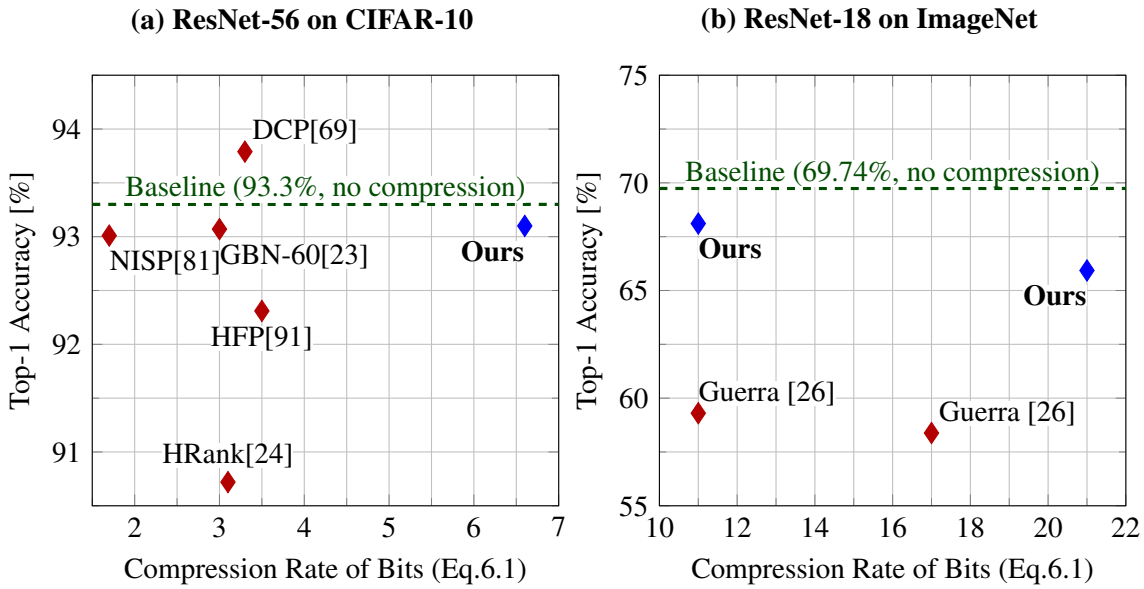

Figure 6.5: Accuracies and compression rates of different reduction methods that reduce the **memory requirements** (see Eq. 6.1) of ResNet-56 and ResNet-18 on CIFAR-10 and ImageNet.

ResNet-20 on CIFAR-10

Table 6.1: Top-1 accuracies and compression rates (CR) regarding the number of bits, the number of bit operations, the bandwidth, and the maximum storage cost of the activations using ResNet-20 on CIFAR-10. Results marked with '-' are not reported by the authors.

Method	CR Mem	CR Bandwidth	CR Max-Storage	CR Bit-Ops	Top-1 %
Baseline	-	-	-	-	92.30
TQT [18]	16	8	8	-	88.71
DQ [77]	15	8	-	-	88.71
DQ [77]	15	-	8	-	88.77
Ours	16	8	8	117	**90.03**
Ours	17.3	9.2	8.5	150	**89.93**

use their pruning method to further reduce the quantized model. In contrast, our Holistic Reduction significantly improves the Top-1 accuracy by almost 9% (68.11% vs. 59.30%) with a bit compression rate of 11. In the second experiment, our approach improves both the compression rate (21 vs. 17) and the Top-1 accuracy (65.93% vs. 58.38%) compared to Guerra *et al.* [26].

6.4.3 Holistic Reduction using all Metrics

In this section, we reduce all four complexity metrics: the number of bits, the number of bit operations, the bandwidth, and the maximum storage cost of the activations. Therefore, Table 6.1 shows the reduction results of ResNet-20 on CIFAR-10. ResNet-20 is rather small (1048KB memory requirement) and consequently comparatively hard to reduce [77]. In our first experiment, we set the target complexities so that the reduction is at least equal to TQT [18] and DQ [77]. Here, we are able to improve the Top-1 accuracy by more than 1% (90.03% vs. 88.71%). In addition, we provide a compression rate of 117 regarding the number of bit operations. In our second experiment, we provide both higher compression rates and a higher Top-1 accuracy compared to TQT and DQ.

6.4.4 Ablation Study on the Holistic Approach

We claim that our holistic approach simultaneously learns both the width and the bit size of each layer so that the target complexity - which is defined by the four complexity criteria - is not exceeded. Thus, the target size is fulfilled without the need to determine in advance how the overall reduction rate is distributed among pruning and quantization. To verify our claim, we compare our holistic approach with five configurations that first prune and then quantize based on predefined pruning and quantization rates, please see Figure 6.1. For example, a compression rate of 50 can be achieved by reducing the average bit size of the parameters by 8 (which corresponds to an average bit size of 4 bits) and pruning the number of parameters by a factor of 6.3. The corresponding result is shown in the third column from the left in Figure 6.1. As noticeable, the predefined configurations differ significantly in their Top-1 accuracies (93% vs. 90.3%). Furthermore, none of the predefined configurations that sequentially perform pruning and quantization achieves the same accuracy as our Holistic Reduction approach (93% vs. 93.7%).

6.4.5 Visualization of VGG7 and ResNet-18

This section analyzes the learned bit sizes and layer widths of VGG7 (CIFAR-10) and ResNet-18 (ImageNet). On the one hand, Figure 6.6 visualizes the result of our Holistic Reduction applied to **VGG7** with a reduction rate of 98% in the **number of bits** (see Equation 6.1). This corresponds to a compression rate of 50. The reduced model yields a test accuracy of 93.51%, which is less than 1% below the baseline accuracy of 94.39%.

The topmost diagram shows the proportional layer sizes of the original model. Here, a proportional layer size of 0.6 means that 60% of the total memory requirement of the original model is allocated to the respective layer. As can be seen, VGG7 consists of six convolutional and two fully-connected layers, with most of the parameters being located in the layers with index conv6 and fc7.

The second diagram shows the final reduction rates after training. Here, a reduction rate of 90% means that the memory requirement of the respective layer is reduced by 90% (by both pruning and quantization). As noticeable, the layers conv6 and fc7 are reduced the most, which is reasonable, as they are the layers with the highest capacity. Especially fc7 is highly over-parameterized and can therefore be reduced by more than 99%.

The third and fourth figures show how the overall reduction rate is distributed among pruning and quantization. Since the target reduction is greater than 90%, the bit sizes are initialized with 4 bits. According to Equation 6.6, a trade-off between the learning task and the reduction loss must be weighed for each layer: The learning task takes advantage of increasing bit sizes, while the pruning loss is reduced by decreasing bit sizes. This trade-off is also evident here. On the one hand, the bit size of the small input layer is increased to 6-bit. On the other hand, the bit sizes of the large layers are reduced to 3-bit or even 2-bit. This is a reasonable result, as it has often been reported that both the input and output layer are more sensitive towards quantization and therefore require higher bit sizes [14]. Furthermore, one can observe that the pruning rates are rather small in combination with low bit sizes. However, for the highly over-parameterized layers (e.g. fc7), the pruning rate significantly increases to support the reduction of bits.

On the other hand, Figure 6.7 visualizes the result of our Holistic Reduction applied to **ResNet-18** with a reduction rate of 95.2% in the **number of bits** (see Equation 6.1). This is equivalent to a compression rate of 21. The reduced model yields a test accuracy of 65.93%. The corresponding experiment is also shown in Figure 6.5 (b). ResNet-18 consists of 18 layers whereas 3 shortcut connections have computational layers (resulting in a total number of 21 layers).

As in the case of VGG7, the largest layers are reduced the most as they offer more capacity. For ResNet-18, these are the layers with indexes 17, 19, and 20. Furthermore, our approach learns layer-specific bit sizes so that the first and the last layer receive more capacity. Regarding the pruning rates, it seems that the small layers in the front half of the network are mainly reduced by quantization. As the layer size increases, pruning is increasingly involved to further reduce the layer complexities. Thus, the largest layers are reduced up to 98%.

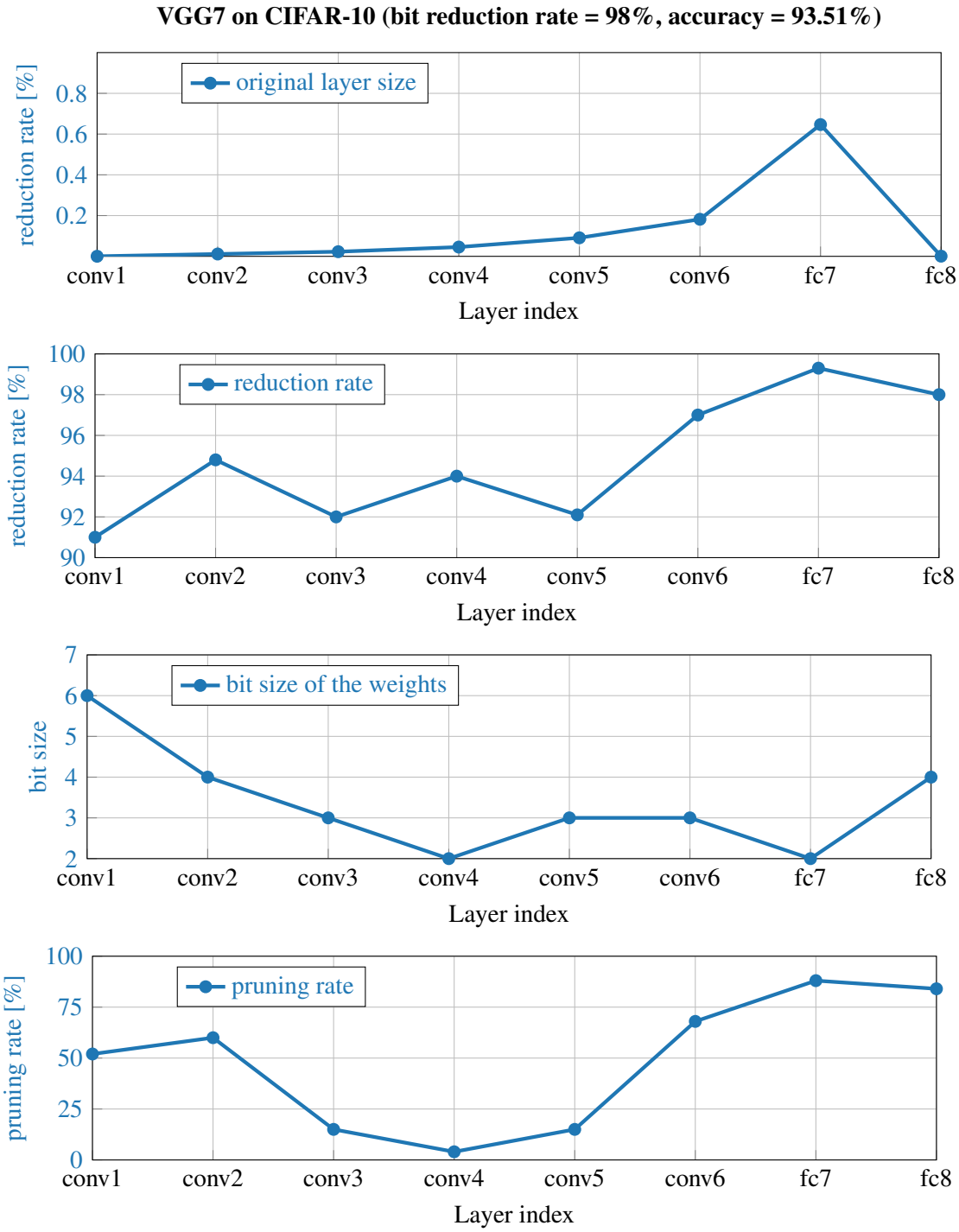

Figure 6.6: Our Holistic Reduction approach applied to VGG7 with a bit compression rate of 50 (which corresponds to 98% bit reduction). The plots visualize the layer-wise reduction rates, pruning rates, and bit sizes after training. Second plot: our approach yields the highest bit reduction rates for proportionally large layers (fc7 with 99.3% bit reduction rate). Third plot: our approach learns layer-specific bit sizes such that the first and the last layer have a higher bit-precision (4-6 bits) compared to intermediate layers (2-3 bits).

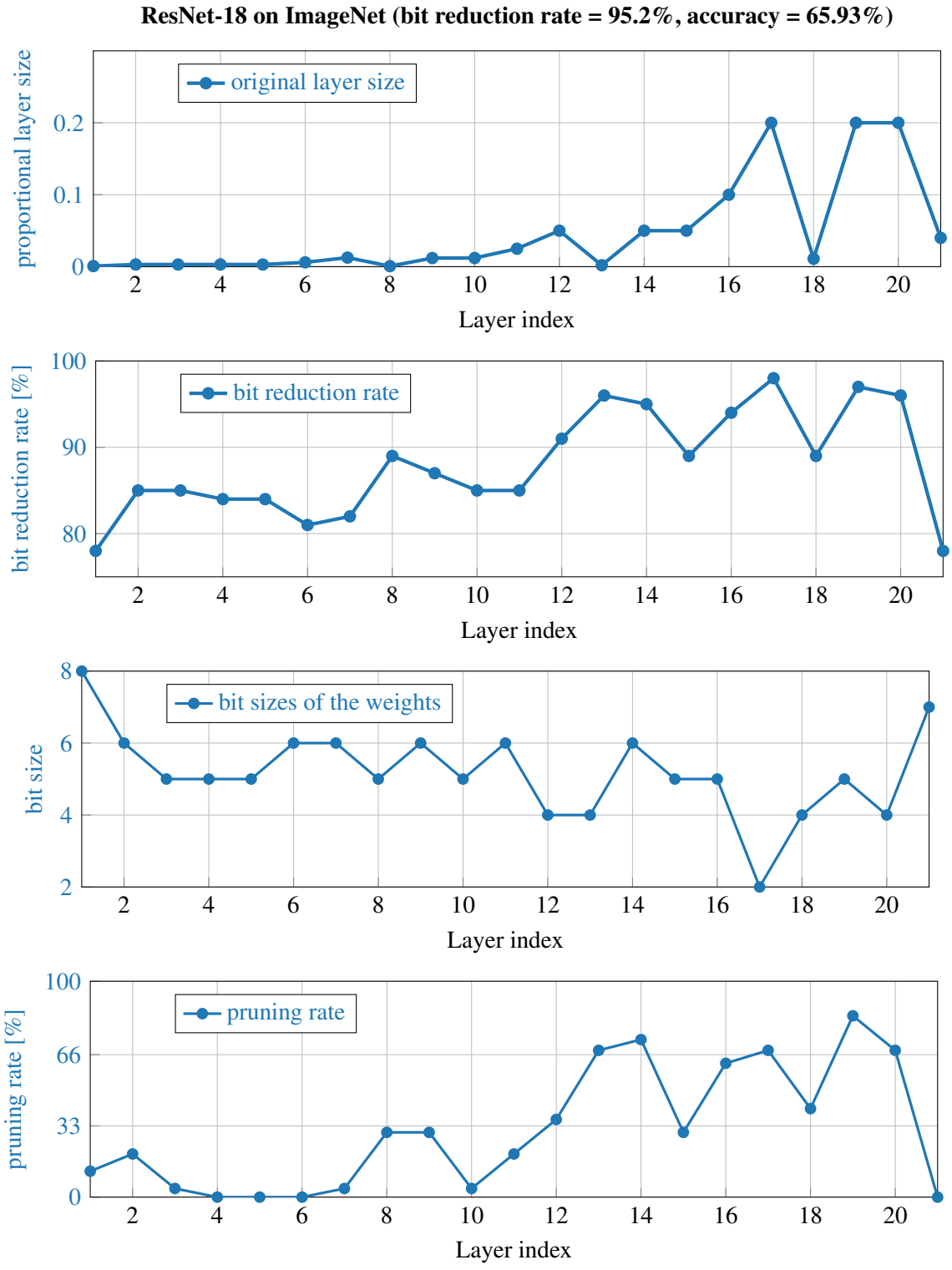

Figure 6.7: Our Holistic Reduction approach applied to ResNet-18 with a compression rate of 21 (which corresponds to 95.2% bit reduction), the result is shown in Figure 6.5. The plots visualize the layer-wise reduction rates, pruning rates, and bit sizes after training. Second plot: our approach yields the highest bit reduction rates for proportionally large layers (layer 17 with 98% bit reduction). Third plot: our approach learns layer-specific bit sizes such that the first and the last layer have higher bit sizes compared to intermediate layers. Fourth plot: the individual pruning rate is low for the first layers and rises with increasing layer size.

6.5 Related Work

Deep neural networks are usually over-parameterized after training and have a high model complexity. When deployed on embedded or mobile devices, both the memory and computational complexity of trained networks must be significantly reduced. Various approaches are available for this purpose. For the proposed method, filter pruning and fixed-point quantization are especially important and thus described in the following.

6.5.1 Filter Pruning

Pruning removes redundant network connections by setting weight values to zero. However, pruning single weights is rather inefficient since it has no direct benefit on the tensor sizes of both the weights and activations and thus leads to unstructured sparsity. In contrast, methods that achieve structured sparsity by removing whole filters and neurons from the network architecture are called filter pruning. These methods can be separated into two categories: saliency-based pruning and sparsity learning.

Saliency-based pruning methods investigate norms to calculate an importance score for each filter. The higher the saliency score the more important the filter is considered to be for calculating the correct network output. Subsequently, filters with the lowest saliency scores are deleted and the remaining ones are retrained. Saliency scores can either be determined by analyzing the magnitudes of the weights and activations [65, 67, 68], the discriminative power of individual filters [69], the reconstruction error of the classification output by setting single filters to zero [81], or the change in the loss function caused by setting single filters to zero [23].

Sparsity learning induces sparsity constraints during the training of deep neural networks. Such sparsity constraints can either be based on the L^0-norm to penalize incoming and outgoing filter connections [89], the LASSO regression to prune filters by minimizing the reconstruction error of the output feature maps [72], the L^1-norm, applied to either the scaling factors of the batch-normalization layers or the sum of the absolute values of the filter weights [76], or on pruning layers that use additional parameters to prune single weights or filters during each forward pass [75]. However, the pruning layers proposed in [75] are located in front of the batch-normalization layers, which reactivate the pruned channels. In [91], a training procedure was proposed that reduces the number of both the parameters and multiplications to a given quantity.

6.5.2 Fixed-Point Quantization

Quantization-aware training methods integrate quantization constraints into the training of deep neural networks. Depending on the type of these quantization constraints, the methods subdivide into two categories: hard and soft quantization.

Hard quantization methods quantize the network weights during each forward pass. During the backward pass, the straight-through estimator is used to approximate the local gradient of the rounding function [53]. This approach became popular with BinaryConnect, which quantized weights to ± 1 [46]. Extensions investigated ternary-valued weights and a real-valued scaling factor [36], as well as asymmetrical weights with two independent scaling factors [49]. Recently, the focus has been shifted to essentially improve the usability of deep networks on embedded devices. These modifications include uniform quantization functions to enable integer domains [13, 55], symmetric and uniform quantization to restrict zero-points to 0 [50, 55], per-tensor quantization to share the same scaling factor across all weights or activations in a tensor[13, 18], and power-of-two step-sizes to enable bit-shift operations [13, 18].

Soft quantization methods use floating-point parameters during training but simultaneously promote parameter distributions that are well-qualified for post-quantization. Soft-quantization methods are either based on Bayesian methods that learn a posterior distribution over the weights by approximating a sparsity-inducing prior [38], Gaussian priors that result in multi-modal fixed-point weights [44, 64], or regularization terms that minimize the quantization error during training [63].

6.5.3 Pruning & Quantization

Han *et al.* combined weight pruning, weight sharing, and Huffman coding to compress the memory footprint of deep neural networks [20]. Tung and Mori proposed a similar approach but performed weight pruning and quantization in parallel [92]. Paupamah *et al.* iteratively pruned the weights with the smallest absolute values in order to perform a per-channel quantization afterwards [25]. However, pruning single weights leads to unstructured sparsity [20, 25, 92], and quantizing using clustering or look-up tables results in time-consuming data accesses during each forward pass [20]. Furthermore, Guerra *et al.* proposed a selection strategy for pruning filters that have already been quantized before [26]. To determine the layer-wise pruning rates, they apply Bayesian optimization. However, since both pruning and quantization aim at reducing the number of required bits, both approaches should be combined in the same optimization problem. Another Bayesian optimization approach for combining pruning and quantization was introduced by Achterhold *et al.* [38]. Therein, Achterhold *et al.* introduced a quantizing prior to train deep neural networks with multi-modal weight distributions. Furthermore, weights with comparatively high variance were set to zero to further reduce complexity.

Kwon *et al.* presented an encryption to decode weight matrices with unstructured sparsity through XOR-gates [102]. However, XOR-gates require very specialized hardware and are limited to binary weights. Recently, Wang *et al.* combined the variational information bottleneck approach and mixed-bit quantization to simultaneously prune and quantize deep neural networks [93]. Furthermore, Wang *et al.* proposed a design methodology for an

efficient network architecture by training an accuracy predictor for both full-precision and quantized deep neural networks [94]. Afterwards, an evolutionary search is performed to find a suitable architecture with quantized weights based on certain hardware constraints.

6.6 Conclusion

In this chapter, we proposed Holistic Reduction, an approach for reducing the complexity of deep neural networks by an efficient combination of filter pruning and fixed-point quantization. More precisely, we presented the first joint pruning and quantization approach that allows defining maximum values for all important complexity metrics: the memory requirement, the computational effort that is based on the number of required bit operations, the bandwidth, and the maximum storage cost of the activations. The maximum values (i.e. the target complexity) can be derived directly from the target device or specified as desired. Based on this, the architecture of the network as well as the bit sizes of both the weights and activations are reduced so that the target complexity is fulfilled. The final model is highly efficient, runs without batch-normalization layers, and has all weights and activations in fixed-point representation with per-tensor decimal points. Compared to related work, our approach has several advantages: the resources available are distributed automatically so that a global solution is found, it can be easily extended to new complexity metrics, and it fits seamlessly into the common training procedure of deep neural networks. In various experiments, we showed state-of-the-art performance on different data sets using several deep learning architectures. Especially for very high reduction rates, our approach achieves excellent performance and reduces the number of required bit operations of ResNet-18 on ImageNet by 55 with no significant loss in accuracy ($\approx 0.5\%$). Furthermore, we examined the learned bit sizes and layer widths and made reasonable observations: the critical input and output layers receive more capacity (both by more channels and higher bit sizes), and the reduction rates of the intermediate layers depend on their respective influence on the target complexity.

Chapter 7

Conclusion

In this thesis, we proposed several approaches to reduce the complexity of deep neural networks. In general, the field of model reduction includes all approaches that are capable of reducing the memory or computational complexity of trained networks. These approaches can be divided into several categories and subcategories, an overview is visualized in Figure 1.1. On the one hand, pruning, factorization, and network distillation reduce the number of parameters and required multiplications by removing redundant network connections. On the other hand, quantization reduces the bit sizes of operands and operations, immediately reducing memory complexity. The benefit of quantization in terms of computational complexity, in contrast, depends on the available hardware resources. Here, a dedicated fixed-point quantization can significantly reduce computation time, area cost, and energy consumption. We made contributions to both pruning and quantization and also provided a particularly efficient combination of both approaches.

In Section 4.2, we addressed the problem of integrating fixed-point constraints into soft quantization approaches. In contrast to hard quantization, soft quantization methods use floating-point parameters during the training but simultaneously promote posterior distributions of the parameters that are well qualified for post quantization. However, previous soft quantization approaches had several drawbacks: they mainly ignored the fixed-point constraint of power-of-two scaling factors, completely neglected the integration of batch-normalization layers, and were often difficult to implement. Thus, we first proposed an approach to train deep neural networks with multi-modal weight distributions and minimal quantization error. With each mode corresponding to a certain fixed-point number, the weights can be quantized using fixed-point arithmetic after training with no significant loss in accuracy. Our approach involves a reduction loss that can be integrated into the common training procedure of deep neural networks with very little implementation effort. In various experiments, our soft quantization approach achieved excellent performance even with low-bit weights.

In Section 4.3, we extended our work on fixed-point quantization and proposed the first soft quantization approach that takes into account the distribution of the parameters of the batch-normalization layers. Here, minimizing the reduction loss promotes distributions of the parameters that resemble a multi-modal distribution after folding the

batch-normalization layers into the preceding convolutional or fully-connected layers. As a result, the batch-normalization layers can be integrated into the fixed-point representation of the preceding layers with so significant loss in accuracy. Furthermore, we used a discrete ReLU activation function that quantizes the network activations during each forward pass. Thus, the trained networks can be evaluated using pure fixed-point arithmetic after training. In this way, we have provided a useful alternative to related hard quantization methods that have significantly higher implementation effort with comparable performance.

Furthermore, in Section 5, we proposed a novel filter pruning method that reduces the memory and computational complexity of a deep neural network to a given target size. Here, our main contribution is the first filter pruning method that allows to directly specify accurate maximum values for the most important complexity metrics: the number of parameters and required multiplications. Based on these two maximum values, the resources available are distributed across the individual layers by pruning filters and neurons from the network architecture. Thus, our major motivation was as follows. The users of deep neural networks usually do not care about the exact size of single layers or how the resources available are distributed over the network depth. However, the main issue is that the final model complexity must not exceed the limitations imposed by the target hardware. On the one hand, the number of model parameters is strictly limited by the amount of storage the network can allocate on the target device. On the other hand, the number of required multiplications results from the available computation units and the capability to parallelize computations. As in the case of our fixed-point quantization approaches, the network is reduced by using an additional reduction loss during training. Before each update step, the reduction loss calculates the difference between the current model size and the target size, in which the current model size can be reduced by pruning filters and neurons via the channel-wise affine transformations of the batch-normalization layers. Thus, a global solution can be found and no additional variables are needed. In various experiments, our filter pruning approach achieved state-of-the-art performance and outperformed related work especially for large pruning rates.

Finally, in Section 6, we proposed a combination of filter pruning and fixed-point quantization that is capable of training very efficient deep neural networks. First, we proposed a novel reduction loss consisting of four essential metrics that serve as a measure of complexity: the memory requirement, the computational cost defined by the number of bit operations, the bandwidth, and the maximum storage cost of the activations. Setting a maximum value for each complexity metric gives a comprehensive description of the target hardware and the reduction loss becomes an indicator of the difference between the actual model complexity and the target complexity. Based on this, we proposed custom layers that either replace or extend the original layers and enable to reduce the reduction loss during training. Here, pruning layers reduce the layer widths and fixed-point quantization layers reduce the bit sizes of both the weights and activations. Thus, we benefit from all the optimization techniques specially designed for training deep neural networks, such

as momentum, running averages, data augmentation, and learning rate schedules. In various experiments, we showed new state-of-the-art performance on different data sets and architectures. We also showed the benefit of our joint pruning and quantization approach over iterative procedures.

For future work, we plan to enhance our holistic reduction approach. It has been shown that a joint combination of filter pruning and fixed-point quantization can significantly reduce the complexity of deep neural networks. On the one hand, fixed-point quantization reduces the bit sizes of both the weights and activations. On the other hand, filter pruning reduces the number of channels in each layer. The very next step is to also reduce the network depth, i.e. the number of layers, by a novel layer pruning method. Combined with our already presented pruning and quantization layers, the entire network architecture could be learned according to the capacity of the target device. This would combine the field of model reduction with that of neural architecture search. Thus, one could train an over-parameterized floating-point model, consisting of many wide layers, and subsequently retrain and reduce the network depth, the layer widths, and the bit precision to find both an efficient and powerful architecture.

Furthermore, we want to extend our soft quantization approach towards weight clustering. In Section 4.2, we proposed a reduction loss that can be used to train deep neural networks with minimal quantization error. In this way, the weights cluster around predefined fixed-point numbers. One intuitive modification therefore involves learning the cluster centers as well. This can be done by deriving the reduction loss with respect to the cluster centers, which can then be updated in the direction of their negative gradients. Thus, an intelligent clustering algorithm can be developed that is easy to implement and learns both appropriate cluster centers and weight values. The number of clusters as well as additional fixed-point constraints can furthermore be specified by the user according to the target device.

List of Figures

List of Tables

Bibliography

[1] D. Kahneman, *Thinking, Fast and Slow*. Penguin, 2011.

[2] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[4] B. Scholkopf, "Causality for machine learning," 2019.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.

[6] M. Kumar and S. Dargan, "A survey of deep learning and its applications: A new paradigm to machine learning," *Archives of Computational Methods in Engineering*, 06 2019.

[7] D. Feng, C. Haase-Schuetz, L. Rosenbaum, H. Hertlein, F. Duffhauss, C. Gläser, W. Wiesbeck, and K. Dietmayer, "Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, pp. 1341–1360, 2021.

[8] B. Mateen, J. Liley, A. Denniston, C. Holmes, and S. Vollmer, "Improving the quality of machine learning in health applications and clinical research," *Nature Machine Intelligence*, Oct. 2020.

[9] A. Keshavarzi Arshadi, J. Webb, M. Salem, E. Cruz, S. Calad-Thomson, N. Ghadirian, J. Collins, E. Diez-Cecilia, B. Kelly, H. Goodarzi, and J. S. Yuan, "Artificial intelligence for covid-19 drug discovery and vaccine development," *Frontiers in Artificial Intelligence*, vol. 3, p. 65, 2020.

[10] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations (ICLR)*, 2019.

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

[12] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

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

[14] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[15] M. Harris, "Mixed-precision programming with cuda 8," *"developer.nvidia.com"*, Oct. 2016.

[16] X. D. Chen, X. Hu, H. Zhou, and N. Xu, "Fxpnet: Training a deep convolutional neural network in fixed-point representation," *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2494–2501, 2017.

[17] L. Mauch and B. Yang, "Least-squares based layerwise pruning of convolutional neural networks," in *2018 IEEE Statistical Signal Processing Workshop, SSP 2018, Freiburg im Breisgau, Germany, June 10-13, 2018*, 2018, pp. 60–64.

[18] S. R. Jain, A. Gural, M. Wu, and C. H. Dick, "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks," *arXiv preprint arXiv:1903.08066*, 2019.

[19] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015.

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

[21] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arxiv*, 10 2017.

[22] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[23] Z. You, K. Yan, J. Ye, M. Ma, and P. Wang, "Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks," in *Advances in Neural Information Processing Systems 32*, 2019.

[24] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, and L. Shao, "Hrank: Filter pruning using high-rank feature map," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[25] K. Paupamah, S. James, and R. Klein, "Quantisation and pruning for neural network compression and regularisation," in *IEEE International SAUPEC/RobMech/PRASA Conference*, 2020.

[26] L. Guerra, B. Zhuang, I. Reid, and T. Drummond, "Automatic pruning for quantized neural networks," *arXiv preprint arXiv:2002.00523*, 2020.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[28] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[29] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. Madison, WI, USA: Omnipress, 2010, p. 807–814.

[30] M. 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)*, Feb 2014, pp. 10–14.

[31] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*. Morgan and Claypool, 2020.

[32] Y. LeCun and C. Cortes, "Mnist handwritten digit database," 2010.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[34] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[35] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[36] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[38] J. Achterhold, J. M. Koehler, A. Schmeink, and T. Genewein, "Variational network quantization," in *International Conference on Learning Representations (ICLR)*, 2018.

[39] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012.

[40] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[41] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1–9.

[42] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations and Trends in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.

[43] A. Karki, C. Palangotu Keshava, S. Mysore Shivakumar, J. Skow, G. Madhukeshwar Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2019, pp. 137–138.

[44] L. Enderich, F. Timm, L. Rosenbaum, and W. Burgard, "Learning multimodal fixed-point weights using gradient descent," *European Symposium on Artificial Neural Networks*, vol. 27, Mar. 2019.

[45] L. Mauch and B. Yang, "A novel layerwise pruning method for model reduction of fully connected deep neural networks," in *2017 International Conference on Acoustics, Speech and Signal Processing, 2017, New Orleans, LA, USA, March 5-9, 2017*, 2017, pp. 2382–2386.

[46] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[47] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 4107–4115.

[48] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2849–2858.

[49] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," in *International Conference on Learning Representations (ICLR)*, 2017.

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

[51] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems 4*. Morgan-Kaufmann, 1991, pp. 950–957.

[52] P. Yin, S. Zhang, J. Lyu, S. Osher, Y. Qi, and J. Xin, "Binaryrelax: A relaxation approach for training deep neural networks with quantized weights," *SIAM Journal on Imaging Sciences*, vol. 11, 2018.

[53] G. Hinton, "Neural networks for machine learning," 2012, coursera, video lecture.

[54] S. Chen, W. Wang, and S. J. Pan, "Metaquant: Learning to quantize by learning to penetrate non-differentiable quantization," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 3916–3926.

[55] A. Goncharenko, A. Denisov, S. Alyamkin, and E. Terentev, "Fast adjustable threshold for uniform neural network quantization," *International Journal of Computer and Information Engineering*, vol. 13, no. 9, pp. 495–499, 2019.

[56] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 3009–3018.

[57] E. Meller, A. Finkelstein, U. Almog, and M. Grobman, "Same, same but different: Recovering neural network quantization error through weight factorization," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 4486–4495.

[58] R. Banner, Y. Nahshan, and D. Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[59] Y. Li, R. Gong, X. Tan, Y. Yang, P. Hu, Q. Zhang, F. Yu, W. Wang, and S. Gu, "Pushing the limit of post-training quantization by block reconstruction," in *International Conference on Learning Representations (ICLR)*, 2021.

[60] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression." *CoRR*, vol. abs/1702.04008, 2017.

[61] C. Louizos, K. Ullrich, and M. Welling, "Bayesian compression for deep learning," in *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 3288–3298.

[62] A. Zhou, A. Yao, K. Wang, and Y. Chen, "Explicit loss-error-aware quantization for low-bit deep neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[63] Y. Choi, M. El-Khamy, and J. Lee, "Learning low precision deep neural networks through regularization," *CoRR*, vol. abs/1809.00095, 2018.

[64] L. Enderich, F. Timm, and W. Burgard, "Symog: learning symmetric mixture of gaussian modes for improved fixed-point quantization," *Neurocomputing*, 2020.

[65] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *International Conference on Learning Representations (ICLR)*, 2016.

[66] C. Lemaire, A. Achkar, and P. Jodoin, "Structured pruning of neural networks with budget-aware regularization," *CoRR*, vol. abs/1811.09332, 2018.

[67] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[68] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[69] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[70] Y. He, P. Liu, Z. Wang, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," *CoRR*, vol. abs/1811.00250, 2019.

[71] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, 2017.

[72] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[73] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l0 regularization," in *International Conference on Learning Representations (ICLR)*, 2018.

[74] S. Srinivas, A. Subramanya, and R. V. Babu, "Training sparse neural networks," *CoRR*, vol. abs/1611.06694, 2016.

[75] X. Xiao, Z. Wang, and S. Rajasekaran, "Autoprune: Automatic network pruning by regularizing auxiliary parameters," in *Advances in Neural Information Processing Systems 32*, 2019.

[76] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[77] S. Uhlich, L. Mauch, K. Yoshiyama, F. Cardinaux, J. A. Garcia, S. Tiedemann, T. Kemp, and A. Nakamura, "Differentiable quantization of deep neural networks," *arXiv preprint arXiv:1905.11452*, 2019.

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

[79] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian, "Variational convolutional neural network pruning," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[80] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. S. Doermann, "Towards optimal structured cnn pruning via generative adversarial learning," *CoRR*, vol. abs/1903.09291, 2019.

[81] R. Yu, A. Li, C. Chen, J. Lai, V. I. Morariu, X. Han, M. Gao, C. Lin, and L. S. Davis, "NISP: Pruning networks using neuron importance score propagation," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[82] Y. Li, S. Gu, C. Mayer, L. V. Gool, and R. Timofte, "Group sparsity: The hinge between filter pruning and decomposition for network compression," in *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[83] Y. He, Y. Ding, P. Liu, L. Zhu, H. Zhang, and Y. Yang, "Learning filter pruning criteria for deep convolutional neural networks acceleration," in *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[84] S. Guo, Y. Wang, Q. Li, and J. Yan, "Dmcp: Differentiable markov channel pruning for neural networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[85] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605.

[86] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[87] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," *CoRR*, vol. abs/1608.04493, 2016.

[88] K. Persand, A. Anderson, and D. Gregg, "Composition of saliency metrics for pruning with a myopic oracle," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, dec 2020.

[89] W. Pan, H. Dong, and Y. Guo, "Dropneuron: Simplifying the structure of deep neural networks," *arXiv preprint arXiv:1606.07326*, 2016.

[90] Q. Huang, S. K. Zhou, S. You, and U. Neumann, "Learning to prune filters in convolutional neural networks," *CoRR*, vol. abs/1801.07365, 2018.

[91] L. Enderich, F. Timm, and W. Burgard, "Holistic filter pruning for efficient deep neural networks," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2021.

[92] F. Tung and G. Mori, "Clip-q: Deep network compression learning by in-parallel pruning-quantization," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[93] Y. Wang, Y. Lu, and T. Blankevoort, "Differentiable joint pruning and quantization for hardware efficiency," in *European Conference on Computer Vision (ECCV)*, 2020.

[94] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han, "APQ: Joint search for network architecture, pruning and quantization policy," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[95] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[96] J. LIU, Z. XU, R. SHI, R. C. C. Cheung, and H. K. So, "Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers," in *International Conference on Learning Representations (ICLR)*, 2020.

[97] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015.

[98] L. N. Smith and N. Topin, "Super-convergence: Very fast training of residual networks using large learning rates," *CoRR*, vol. abs/1708.07120, 2017.

[99] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[100] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *International Conference on Computer Vision (ICCV)*, 2019.

[101] B. Dai, C. Zhu, B. Guo, and D. Wipf, "Compressing neural networks using the variational information bottleneck," in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, vol. 80, 2018.

[102] S. J. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park, and G.-Y. Wei, "Structured compression by weight encryption for unstructured pruning and quantization," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.