UNIVERSITY OF CALIFORNIA, MERCED

Low-rank Compression of Neural Networks: LC Algorithms and Open-source Implementation

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering & Computer Science

by

Yerlan Idelbayev

Committee in charge:

Professor Miguel Á. Carreira-Perpiñán, Chair
Professor Alberto Cerpa
Professor Sungjin Im

2021

The dissertation of Yerlan Idelbayev is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

(Professor Alberto Cerpa)

(Professor Sungjin Im)

(Professor Miguel Á. Carreira-Perpiñán, Chair)

University of California, Merced

2021

DEDICATION

To my family: Russell, Gulnar, and Rabiga.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not be possible without many wonderful people I have met throughout my life, and I am thankful for knowing them.

First and foremost, I want to thank my advisor Dr. Miguel Á. Carreira-Perpiñán for all his advice, guidance, and examples on how to be an excellent scientist. I want to thank the committee members: Dr. Alberto Cerpa and Dr. Sungjin Im for their feedback on my research during my years at UC Merced. Their help and wisdom were always appreciated.

My graduate journey would not be possible without the aspiring influence of my teachers during the undergraduate studies. I want to thank Maksat Maratov, Victor Dmitriyev, and Dr. Daniyar Turmukhambetov for all their lessons: many of my programming skills stem from their classes in computer science.

My friends from Kazakhstan, Nursultan, Yesdaulet, and Shalkar: I am grateful for sharing this journey with you. I will always cherish our time together in Kazakhstan and in the US. I want to acknowledge my friends from the US as well. I want to thank Teresa and Eric from Davis for all their help in getting to know the culture and language of this, new for me, country. Special thanks go to Alma and Eldar from San Diego: your advice on grad school, life, and everything else was most helpful and tremendously impacted my career. This list would not be complete without mentioning Aleksey Sokolov, Nurlybek Kasimov, and the members of the Kazakh community in Davis, San Diego, and Merced: our game nights, outdoor matches and activities were full of fun.

Lastly, I want to thank my family. I am forever indebted to my mother, Rabiga, for her enormous effort in raising me in a loving and inspiring environment and fostering my curiosity and interest in sciences. My deepest appreciation goes to my wife, Gulnar. Without her wisdom and support beside me, this dissertation would not be possible. I also thank my uncle Ruslan Tlemisov for his encouragement and help on this journey. During the last year of my PhD we have had our son Russell, who became the center of our universe and major source of happiness. I acknowledge the effort of our extended family in helping and advising on nurturing Russell: Elvira, Sabina, and Gulnar's parents Zaure and Batyrkhan, thank you!

VITA

2013 Bachelor of Science in Information Systems *with distinction*, International Information Technology University, Almaty, Kazakhstan

2016 Master of Science in Computer Science, University of California San Diego

2020 Outstanding Teaching Award for valuable contributions to teaching and pedagogy at University of California, Merced

2022 Ph. D. in Electrical Engineering and Computer Science, University of California, Merced

PUBLICATIONS

Yerlan Idelbayev, Arman Zharmagambetov, Magzhan Gabidolla and Miguel Á. Carreira-Perpiñán: "Faster neural net inference via forests of sparse oblique decision trees," *in submission (2021).*

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "LC: A flexible, extensible open-source toolkit for model compression," in *Conference on Information and Knowledge Management (CIKM 2021)*, resource paper.

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "More general and effective model compression via an additive combination of compressions", in *32nd European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD 2021).*

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "Beyond FLOPs in low-rank compression of neural networks: optimizing device-specific inference runtime," in *IEEE International Conference on Image Processing (ICIP 2021)*, pp. 2843–2847.

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "An empirical comparison of quantization, pruning and low-rank neural network compression using the LC toolkit," in *International Joint Conference on Neural Networks (IJCNN 2021).*

Yerlan Idelbayev, Pavlo Molchanov, Maying Shen, Hongxu Yin, Miguel Á. Carreira-Perpiñán and Jose M. Alvarez: "Optimal quantization using scaled codebook," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2021)*, pp. 12095–12104.

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "Optimal selection of matrix shape and decomposition scheme for neural network compression," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2021)*, pp. 3250–3254.

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "Neural network compression via additive combination of reshaped, low-rank matrices," in *Data Compression Conference (DCC 2021)*, pp. 243–252.

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "Low-rank compression of neural nets: learning the rank of each layer," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020)*, pp. 8046–8056.

Elad Eban, Yair Movshovitz-Attias, Hao Wu, Mark Sandler, Andrew Poon, Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán: "Structured multi-hashing for model compression," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2020)*, pp. 11900–11909.

Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev: "'Learning-compression' algorithms for neural net pruning," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2018)*, pp. 8532–8541.

Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev: "Model Compression as Constrained Optimization, with Application to Neural Nets. Part II: quantization," Unpublished manuscript, July 13, 2017, arXiv:1707.04319

Chen Zhang, Yerlan Idelbayev, Nicholas Roberts, Yiwen Tao, Yashwanth Nannapaneni, Brendan M. Duggan, Jie Min, Eugene C. Lin, Erik C. Gerwick, Garrison W. Cottrell and William H. Gerwick: "Small Molecule Accurate Recognition Technology (SMART) to Enhance Natural Products Research," in *Scientific Reports 7, 14243 (2017)*

ABSTRACT OF THE DISSERTATION

Low-rank Compression of Neural Networks: LC Algorithms and Open-source Implementation

by

Yerlan Idelbayev
Doctor of Philosophy in Electrical Engineering & Computer Science

University of California Merced, 2021

Professor Miguel Á. Carreira-Perpiñán, Chair

Neural networks have gained widespread use in many machine learning tasks due to their state-of-the-art performance. However, the cost of this progress lies in the ever-increasing sizes and computational demands of the resulting models. As such, the neural network compression, the process of reducing the size, power consumption, or any other cost of interest of the model, has become an important practical step when deploying the trained models to perform inference tasks.

In this dissertation, we explore a particular compression mechanism — the low-rank decomposition — and its extensions for the purposes of neural network compression. We study important aspects of the low-rank compression: how to select the decomposition ranks across the layers, how to choose best decomposition shapes for non-matrix weights among a number of options, and how to adapt the low-rank scheme to target the inference speed. Computationally, these are hard problems involving integer variables (ranks, decomposition shapes) and continuous variables (weights), as well as nonlinear loss and constraints.

As we show over the course of this dissertation, all these problems admit suitable formulations that can be efficiently solved using the recently proposed *learning-compression algorithm*. The algorithm relies on the alternation of two optimization steps: the step over the neural network parameters, the L step, and the step over the compression parameters, the C step. Once we formulate the compression problems, we show how the L and C steps are derived. Each step can be solved efficiently: the L step is solved by stochastic gradient descent, and the C step relies on singular value decomposition. We demonstrate the effectiveness of the proposed compression schemes and the corresponding algorithms on multiple networks and datasets.

Finally, we discuss the resulting general neural network compression toolkit that encompasses all compression schemes presented in this dissertation and many others. The toolkit is designed to be flexible and extensible, and is released under the open-source license.

Chapter 1

Introduction

Neural networks have established state-of-the-art performance nearly in every machine learning task, and currently, are the method of choice for problems in the fields of image, audio, and video classification, natural language processing, speech to text and text to speech processing, and others. With such a wide application space, the neural networks have become an important practical tool in day-to-day activities. For instance, you can find dozens of neural networks deployed around you and your devices as image and video enhancers in cameras and smartphones, as voice-to-text modules in virtual keyboards, or as personal assistants in your favorite email clients.

The improvement in the performance of the neural networks is typically attributed to the following four factors: 1) availability of large-scale datasets such as ImageNet [108] or Microsoft COCO [87] with high quality labeling information; 2) availability of efficient hardware such as graphical processing units (GPUs) and custom build accelerators like tensor processing units (TPUs) [68], which allow to speed up the training of the neural networks; 3) availability of open-source software frameworks such as Caffe [67], Theano [115], TensorFlow [1], PyTorch [103], which ease the burden of training and experimentation; and finally 4) research on better training techniques, such as novel initialization [34, 44] or normalization [63] methods, and the sheer amount of accumulated empirical knowledge on typical settings of the algorithms, i.e., the training recipes. These factors in combination allow us to train ever-increasing neural networks with better performances.

We can illustrate the evolution of neural network designs and performances on the large scale image recognition task of Russakovsky et al. [108]. This challenge, also known as ImageNet-2012 or ILSVRC2012, asks to create a machine learning model able to classify a color image into one of the 1 000 classes. Starting from the seminal paper of Krizhevsky et al. [75], in which authors proposed a neural network (now called AlexNet) achieving 81.8% top-5 accuracy, neural networks started to dominate the challenge at the cost of increasing complexity and computational demand. The winner of the year 2012, the AlexNet, had only 8 layer and required 727 MFLOPs to classify a single image. The winners of following years have more layers and require more computation: for example, the winner of 2014 had 22 layers and 2 GFLOPs [112], the winner of 2015 had 154 layers and 11 GFLOPs [45], etc. On the bottom of Figure 1.1 we show a summary of the performances of different architectures on the ImageNet 2012 task. On the top of Figure 1.1, we show a historic overview of the number of trainable parameters in prominent neural networks.

As you can see from Figure 1.1, improvements in the network accuracy come from training larger and more demanding neural networks. Yet, the trend of "larger network — better accuracy" comes contrary to the economic and business considerations when these models are actually used (deployed). We can identify two primary deployment scenarios:

- **Cluster deployment**. When deploying a network in a cluster environment, we have access to powerful computers with lots of memory and computational resources. However, running on those powerful machines might be an expensive operation, especially if we are running at scale. Thus we would like to use fewer resources like CPUs and GPUs, virtual machines, and consume less power to save money.

- **Edge deployment**. In this scenario, we deploy trained networks on low-power devices like smartphones that are closest to the end-user. Such devices have stringent constraints in terms of available resources: memory, bandwidth, power, etc., therefore the deployed model can fail to run at all, e.g., if it does not fit into the available RAM.

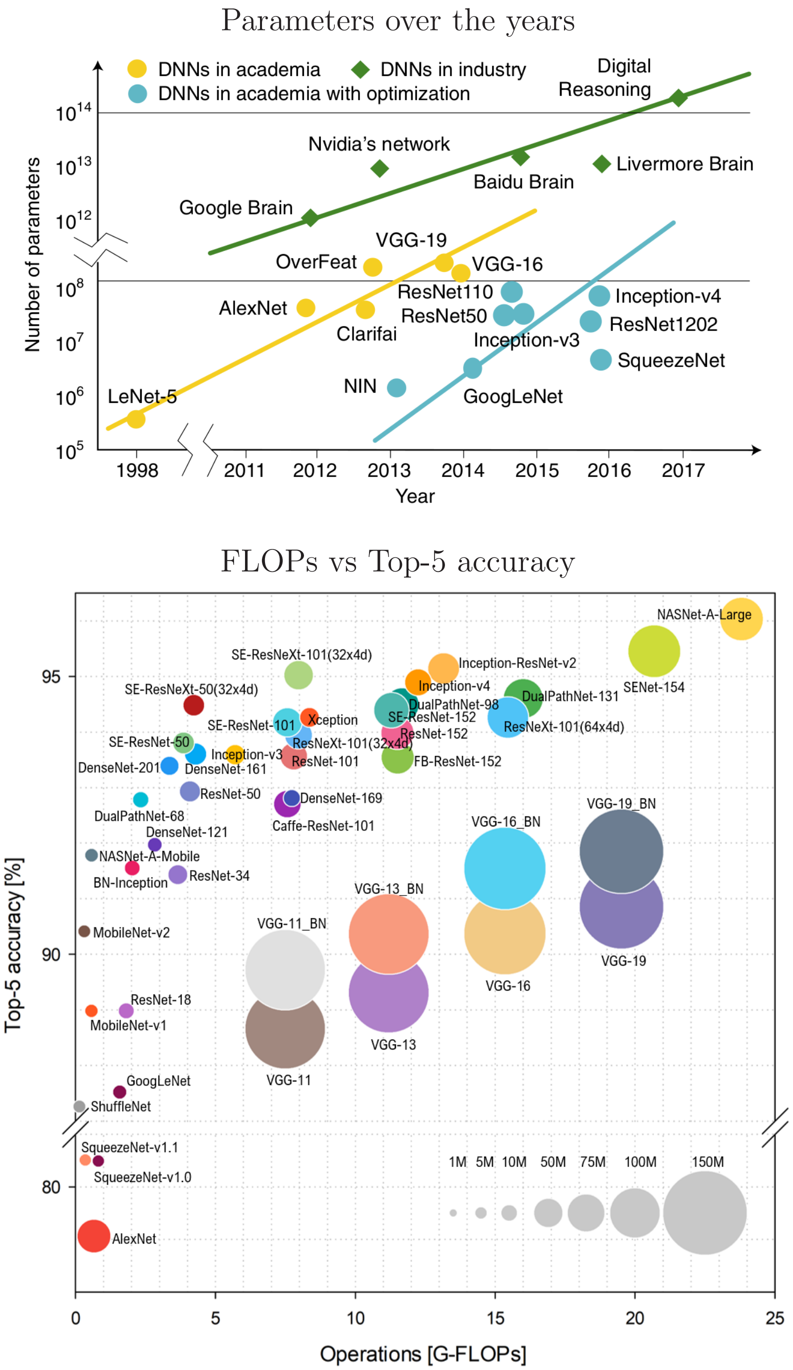


Figure 1.1: Summary of performances of different neural networks on the ImageNet-2012 task. Top: Historic perspective on number of parameters in the leading neural networks over the years (reprinted by permission from Springer Nature Customer Service Centre GmbH: Nature Electronics, Scaling for Edge Inference of Deep Neural Network [125], © 2018). Bottom: FLOPs vs top-5 accuracies for different networks on ILSVRC2012 task (obtained from Bianco et al. [9]).

This leads to *the problem of model compression* — how we can modify the parameters of the model (neural network) so that it requires fewer resources and fits into device constraints (say, cheaper to run or has smaller size) while maintaining the reference accuracy of the model.

Out of many compression strategies that have been developed in the literature, in this dissertation we focus on using the low-rank decompositions. Such a compression scheme has several advantages. Firstly, low-rank methods have a history of usage in the fields of linear algebra, signal processing, and statistics with robust computational routines like singular value decomposition (SVD) and well-tested software packages like BLAS and LAPACK. Secondly, when the neural network weights are compressed using the matrices with appropriately small ranks, it *reduces both the size of the network and the computational requirements needed for the forward pass.* Most importantly, the computational savings are realizable without explicit support from the hardware. Indeed, if the weight matrix $\mathbf{W}$ is of low-rank, it can be seen as a product of matrices $\mathbf{U}\mathbf{V}^T$. The forward pass $\mathbf{W}\mathbf{x}$ through such layer now can be computed as a forward pass through a sequence of two regular layers: first through a layer with weights $\mathbf{V}^T$ and then through a layer with weights $\mathbf{U}$. This hardware friendliness is in stark contrast compared to other compression schemes: e.g., quantized or element-wise pruned models require building a dedicated processor to be efficiently deployed [38].

1.1 Contributions

In this dissertation, we study the low-rank compression of the neural networks and several important extensions to it: device targeted compression and the low-rank compression with joint selection of decomposition shapes. These compression problems are hard since each involves a combinatorial substructure: for instance, these problems involve selecting a rank for each of the K layers in the network, which means the number of different choices we need to make is exponential in K.

As we will review in section 2.1, low-rank decomposition has been used for model compression with different degrees of success, often relying on heuristics or using training algorithms that cannot be generalized. Our approach is different as we 1) formulate the compression task as a well-specified optimization problem; and 2) tackle many forms of low-rank compression using a single framework of the learning-compression algorithm of Carreira-Perpiñán [15], which makes our approach generic and extensible. The resulting algorithms are simple, yet competitive: we achieve similar or exceeding compression ratios when compared to the leading methods from the literature at the same accuracy levels. The results presented in this dissertation have been published in peer-reviewed conferences and now constitute solid baselines for neural network compression problems.

Additionally, we present an open source toolkit written in Python that incorporates all low-rank compressions discussed in this dissertations and many others forms of compressions: quantization, pruning, and combinations of those. This software builds on top of the LC algorithm and is used in all of our experiments and publications. Below we give a summary of every chapter in this dissertation.

- In Chapter 2 we present an extensive review of neural network compression works spanning various compression mechanisms like low-rank compression (section 2.1), pruning (section 2.2), quantization (section 2.3), and their combinations. We additionally present an overview of available model compression software and their limitations in section 2.4.

- In Chapter 3 we review the Learning-Compression algorithm which is the backbone optimization method used in our research. The algorithm allows us to efficiently solve the model compression problems formulated using constrained optimization, which is achieved by separation of the model learning from the model compression.

- In Chapter 4 we apply the LC algorithm to solve the problem of low-rank compression of the neural networks. We show that with a suitable formulation we can jointly learn both weights and the ranks of the neural networks to minimize the cost of interest like total floating point operations (FLOPs) in the resulting model.

- In Chapter 5 we show how our formulation of low-rank compression can be naturally extended to handle device-targeted compression: that is how can we jointly train weights and select ranks for a model so that it runs as fast as possible on a given hardware.

- In Chapter 6 we discuss in depth the application of low rank for compression to convolutional layers, which admit several different forms of low-rank decompositions. We propose to select the optimal decomposition form as part of the optimization and compare it to regular low-rank compression schemes.
- In Chapter 7 we present an open-source software framework based on our research: the LC toolkit. We will discuss the design choices behind the library, and the implemented features that makes library extensible and flexible.
- In Chapter 8 we conclude the dissertation and outline possible future directions for our research.

Chapter 2

Related Work

This chapter presents an extensive overview of various compression mechanisms studied in the literature. Due to the nature of the neural network compression field (with hundreds of publications each year), this review is not an exhaustive overview, and certain methods may be already outdated.

2.1 Low rank and other decompositions

Decompositions of a matrix as a product of lower-rank matrices, including the low-rank and tensor ones, have been thoroughly studied in the fields of linear algebra, statistics, engineering, and have found multitudes of applications. For a thorough review of different tensor decomposition types we refer to Kolda and Bader [71]. In this section we limit our attention to relevant tensor and low-rank decomposition works which are used for neural network compression: either to reduce the number of parameters in the net or to speed up the inference time.

If the weight matrix $\mathbf{W}$ of shape $a \times b$ can be naturally decomposed into a product of r-rank matrices $\mathbf{U}\mathbf{V}^T$, we can store $\mathbf{U}$ and $\mathbf{V}$ separately by using $r \times (a+b)$ floating point values. When r is suitably small, i.e., $r \leq ab/(a+b)$, storing such decomposed weights (instead of the original $\mathbf{W}$) becomes an efficient compression mechanism. The decomposed matrices not only save storage, but also reduce the computational load: instead of computing $\mathbf{y} = \mathbf{W}\mathbf{x}$ during the forward pass of the neural network, we would compute the intermediate product $\mathbf{x}' = \mathbf{V}^T\mathbf{x}$

first, and then compute $\mathbf{y} = \mathbf{U}\mathbf{x}'$. Such chaining reduces the total FLOPs count of the model (see appendix A), and, most importantly, can be efficiently implemented in any neural network and underlying hardware: it is equivalent to having two linear layers with weights $\mathbf{U}$ and $\mathbf{V}^T$ instead of one layer with $\mathbf{W}$.

Generalizations of low-rank decompositions for higher order matrices, i.e., tensors, also have compressing properties and have been widely explored in the neural network compression literature. In Table 2.1 we group and summarize the various types of decompositions used for network compression. Aside from the decomposition type, there are several important characterizations of these works that we review next. In section 2.1.1 we review simple methods which heuristically select the ranks and in section 2.1.2 we review typical heuristics. In section 2.1.3 we review the works that formulate principled optimization problems to train low-rank networks. Finally, in section 2.1.4 we review how matrix decomposition can be extended to convolutional layers.

Before we proceed with the main review of low-rank compression works, we would like to note that some methods use decompositions to create a new architecture rather than using it as a compression tool. These methods decompose the layer, say as $\mathbf{U}\mathbf{V}^T$, and use it as a new layer and train the network from scratch wrt to parameters $\mathbf{U}$ and $\mathbf{V}$ instead of original $\mathbf{W}$. Such an approach was used on image classification problems [84, 117, 129], as well as on language models where extremely large matrices appear in the final layers [18, 109]. Interestingly, now popular depth-wise separable convolutions of Chollet [20], Sandler et al. [110] are a particular version of tensor CP decomposition applied to the weights of a convolutional layer.

Throughout this section we will be denoting the weight matrix or weight tensor interchangeably using bold math symbols (e.g., $\mathbf{W}$), and its decomposition as $\boldsymbol{\Delta}(\boldsymbol{\Theta})$. In case of low-rank decomposition applied to matrix $\mathbf{W}$ we have $\boldsymbol{\Delta}(\boldsymbol{\Theta}) = \mathbf{U}\mathbf{V}^T$ with $\boldsymbol{\Theta} = \{\mathbf{U}, \mathbf{V}\}$.

Type	Specifics	Used in works of
Low rank	scheme 1	Denil et al. [25], [26], Zhang et al. [135], Wen et al. [120], Xu et al. [125], Li and Shi [80]
	scheme 2	Jaderberg et al. [66], [113], Xu et al. [125], Kim et al. [69]
Tensor decompositions	CP	Denton et al. [26], [77]
	Tucker	Kim et al. [70]
	Tensor-Train	Novikov et al. [102] FC layers only, Garipov et al. [33]
	Tensor-Ring	Wang et al. [117]
	Block-Term	Ye et al. [129]
Other		Jaderberg et al. [66] as scheme 1 with filters of rank 1 Ioannou et al. [61], Ioannou et al. [62] decompose with linear combination of different rank matrices

Table 2.1: Summary of different decompositions used for neural network compression.

2.1.1 Early methods

Early methods simply applied a chosen form of decomposition to the pre-trained weights, and then optionally retrained the decomposed weight matrices on the neural network task L by solving the problem of $\min_{\mathbf{\Theta}} L(\mathbf{\Delta}(\mathbf{\Theta}))$. We can characterize these methods by decompositions strategies as *data dependent* and *data independent*.

Data-independent strategies minimize the normed difference between the original weights $\mathbf{W}$ and its decomposition $\boldsymbol{\Delta}(\boldsymbol{\Theta})$:

$$\min_{\boldsymbol{\Theta}} \|\mathbf{W} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|. \tag{2.1}$$

For certain decomposition forms (low-rank for a matrix or Tucker 2 for a tensor) this problem can be solved optimally using Singular Value Decomposition (SVD). Denton et al. [26] and Tai et al. [113] heuristically choose the ranks, initialized the low-rank weight with SVD solution and fine-tuned it on the original task of the network. The work of Denton et al. [26] additionally introduce a scheme where weight tensors are approximated as outer products of rank-1 matrices, for which they use alternating least squares to optimize the corresponding decomposition problem of eq. (2.1). In the works of Jaderberg et al. [66], Lebedev et al. [77] and Novikov et al. [102] the chosen decomposition forms do not have efficient solutions, therefore authors used iterative methods to populate the decomposed models, and did not retrained weights afterwards.

Data-dependent strategies minimize a data-dependent norm of eq. (2.1), for example Denton et al. [26] use the Mahalanobis distance, or minimize the normed difference between responses of original weights $\mathbf{W}$ and its decomposition $\boldsymbol{\Delta}(\boldsymbol{\Theta})$ wrt to datapoints $\mathbf{x}$ in layer-wise fashion:

$$\min_{\boldsymbol{\Theta}} \sum_{\mathbf{x}} \|\mathbf{W}\mathbf{x} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\mathbf{x}\|. \tag{2.2}$$

Denil et al. [25] use low-rank matrix decomposition with $\boldsymbol{\Delta}(\boldsymbol{\Theta}) = \mathbf{U}\mathbf{V}^T$, and solve the eq. (2.1) approximately by computing $\mathbf{U}$ as the kernel ridge regression, $\mathbf{U} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{W}$, for a heuristically chosen kernel matrix $\mathbf{K}$. Jaderberg et al. [66] solved the data-dependent problem of the eq. (2.2) using LBFGS solver, where the initialization come from the solution of the data-independent version of the problem. Zhang et al. [135] recognized that low-rank decomposition version of the eq. (2.2) is a well known reduced rank regression (RRR) problem, which has an optimal solution using the SVD of a certain covariance matrix.

Initial empirical evidence in the literature suggested that finetuning of the decomposed weights wrt original loss of the neural network, i.e., $\min_{\boldsymbol{\Theta}} L(\boldsymbol{\Delta}(\boldsymbol{\Theta}))$, is not easy or does not produce better results, thus after decompositions of the weights, factor matrices remained as is. For example, Zhang et al. [135] write “finetuning is very sensitive to the initialization (given by the approximated model) and the learning rate”, and Jaderberg et al. [66] note that finetuning “does not actually result in better classification accuracy than doing data ℓ_2 reconstruction optimization”. Therefore, initial works restricted finetuning only to a subset of the weights: e.g., Denil et al. [25] finetune only $\mathbf{V}$-matrices of decompositions, keeping $\mathbf{U}$s fixed; Denton et al. [26] finetune only non-decomposed weights. On the other hand, Lebedev et al. [77] were able to finetune the entirety of the decomposed network, though they report that “gradients within the decomposed layers are prone to the gradient explosion” and “proper care should be taken when selecting the learning rates”. Tai et al. [113] were able to finetune low-rank networks, and attribute the success to usage of the batch normalization transformation of Ioffe and Szegedy [63] between the layers of the network.

2.1.2 Heuristic criteria for rank selection

Most of the methods discussed in this section use heuristics to determine the ranks of the decompositions. We call such criteria to be heuristic even though they come as a solution of some optimization problem. Yet, the underlying problem these heuristics are solving completely disregard the task loss L of the neural network and replace it with an easy to solve surrogate function that has no relation to the original L function. Here are some of the most used heuristics:

H1 Zhang et al. [135] proposed to estimate the reduced ranks of the weight matrices by greedily maximizing the accumulated sum of singular values of the matrices, subject to resulting FLOPs of the network is being within a p-proportion, where p is user defined hyper-parameter.

H2 Tai et al. [113], Wen et al. [120] and Xu et al. [125] use a simpler heuristic: choose the rank r_i of the layer i in such way that r_i-rank approximation is within a p-ratio of the norm of the original matrix, i.e., choose the highest rank satisfying $\|\mathbf{UV}\|_F \leq p\|\mathbf{W}\|_F$. For example, Wen et al. [120] and Tai et al. [113] use $p = 0.95$ and Xu et al. [125] use $p = 0.95$ and $p = 0.99$.

H3 Gusak et al. [36], Kim et al. [70] select the ranks by treating the data-independent approximation problem of eq. (2.1) as a Bayesian Matrix Factorization problem and solve it using variational procedure of Nakajima et al. [99]

Heuristics H1 and H2 require a single SVD to compute the ranks estimates, therefore they are fast. Heuristic H3 might take some time depending on the matrix size. Other heuristics used in the literature can be easily extracted from the combinations of above.

2.1.3 Joint optimization of ranks and weights

While a large body of the methods use heuristically determined ranks for decompositions, and then finetune, there are methods that treat low-rank and tensor compression problems as joint optimization over the ranks and weights so that the overall task loss of the network is minimized.

Direct handling of the rank Li and Shi [80] formulate the rank-selection problem jointly with neural network training, minimizing the loss of a network with constraints on the total number of allowed weights and computation, which depends on the rank of each layer. This problem is solved approximately in an alternating manner, where one step optimizes over the weights via SGD, and another step optimizes over the ranks but requires the solution of a mixed-integer program (involving discrete and continuous variables, NP-hard) using the commercial software MOSEK.

Indirect handling of the rank Instead of handling the ranks directly, one line of work is to penalize ranks indirectly. One option for such a penalty is convex relaxation of the rank function — the nuclear norm $\|\mathbf{W}\|_*$. It is defined as a sum of singular values of the matrix, and proven to be a convex envelope of the rank function, i.e., the closest convex lower bound in point-wise sense [31]. The nuclear norm minimization enjoys certain guarantees when used in compressed sensing, i.e., when searching for the lowest-rank matrix that satisfies the partial observations [14, 105]. Methods that use nuclear norm optimize the penalized objective of $\min_{\mathbf{W}} L(\mathbf{W}) + \lambda \|\mathbf{W}\|_*$, and use gradient based optimizers that require running SVD to compute the gradient wrt nuclear norm part. Such formulation has been used to compress single-layer networks in the work of Harchaoui et al. [41], and multi-layer networks in the work of Alvarez and Salzmann [4].

Other low-rank penalties have been explored as well. Wen et al. [120] propose to train a neural network with a "force regularization" penalty R defined per every pair of rows $\mathbf{w}_i, \mathbf{w}_j$ of matrix $\mathbf{W}$:

$$R(\mathbf{W}) = \sum_{i,j} \left\| \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|} - \frac{\mathbf{w}_j}{\|\mathbf{w}_j\|} \right\|.$$

Another line of work that handles ranks indirectly is pruning based approaches applied to the decomposition structure. Most of the decompositions of interest can be seen as a sum of small building blocks $\mathbf{B}_i$: in case of matrices, any r-rank matrix can be seen as a sum of r matrices of rank 1. These methods explicitly rewrite full weight matrices in its complete decomposition form as $\mathbf{W} = \sum_i \alpha_i \mathbf{B}_i$, and jointly train on the task loss L while having ℓ_1 penalty on α_i values. If some of the α_i values get pruned due to the sparsifying properties of ℓ_1 norm, the rank of the $\mathbf{W}$ matrix reduces. Xu et al. [125] adopts such approach for low-rank decompositions, and the work of Kossaifi et al. [72] use it for CP decomposition.

There are several disadvantages of indirectly handling the ranks. First, it is unclear how to extend these methods to include rank-based costs, say, power-consumption of r-rank layer. Without a cost-driven approach, resulting decomposed networks will be suboptimal in comparison to the models that directly minimize the cost of interest. Second, the optimization procedures of those methods are not well suited for training of deep nets. For instance, the nuclear norm based approaches require SVD after every stochastic gradient step, which is an expensive operation. For pruning based approaches, the main difficulty comes in ℓ_1 penalty and stochasticity of the gradient. To determine whether the building block of decomposition has been pruned, its α_i value must be exactly zero, which is impossible when training with SGD. Thus, all of these methods revert to heuristic rank selection (see section 2.1.2) at the end or during the training to determine the ranks of decompositions.

2.1.4 Low-rank parameterization of convolutional layers

Finally, we review how we can apply low-rank matrix parameterization for tensors, particularly for tensors coming from weights of convolutional layers. A convolutional layer with n filters of c channels and $d \times d$ spatial resolution has ncd^2 parameters and naturally forms a tensor of size $n \times c \times d \times d$. We can parametrized it with one of the following low-rank structures which in turn can be implemented as a sequence of convolutions:

Scheme 1 We can view the convolutional weights as a linear layer with shape of $n \times cd^2$ applied to appropriately reshaped volumes of the input. The rank-r approximation then has two linear mappings with weight shapes of $n \times r$ and $r \times cd^2$, which can be efficiently implemented as a sequence of two convolutional layers: first with r filters of shape $c \times d \times d$, and second with n filters of shape $r \times 1 \times 1$. Such parameterization was used in works of Li and Shi [80], Wen et al. [120], Xu et al. [125] and others.

Scheme 2 Alternatively, we can view the convolutional weights as a linear layer with shape of $nd \times cd$ applied to reshaped volumes of the input. For this scheme, an approximation of rank r will have two linear mappings with weight shapes of $nd \times r$ and $r \times cd$, which can be implemented as a sequence of two convolutional layers: first with r filters of $c \times d \times 1$ and second with n filters of $r \times 1 \times d$. This parameterization was used by Jaderberg et al. [66], Tai et al. [113].

Scheme 3 Over the course of our research, we have discovered a third decomposition shape which is a mirror of scheme 1. Instead of reshaping the weights as $n \times cd^2$, we now reshape them as $nd^2 \times c$. A low rank layer in this scheme can be implemented as a sequence of following layers: first with r filters of shape $1 \times 1 \times c$, and second with n filters of $d \times d \times r$.

The aforementioned process of reshaping of a tensor into a matrix is generally known as *matricization or unfolding* of a tensor. In tensor algebra literature various unfolding schemes have been studied, however, only a few of them can be efficiently supported by modern deep-learning frameworks and underlying hardware.

2.2 Pruning and sparsification

The problem of neural network pruning is as old as neural networks themselves. Once an efficient training method using the gradient backpropagation was proposed [107], it was empirically observed that training a larger model is often easier, yet, having a smaller model is more desirable. The primary motivation for initial pruning methods was to achieve smaller and more generalizable models. Indeed, by Occam's razor principle, out of many possible models that perfectly fit the data, the one having the fewest weights usually generalizes better (i.e., does not overfit). Pruning has other advantages: it allows us to get an insight into the network's learning (e.g., which features are important), but most importantly it reduces the size and computational requirements of the model, which makes it an efficient compression mechanism.

We can apply pruning on the level of individual weights, which is known as *unstructured pruning*, or on the level of neurons and filters — *structured pruning*. Very sparse models can be achieved using the unstructured pruning, but the efficient deployment of such models requires the support of sparse matrix-vector multiplications. In comparison, the structured pruning methods achieve moderate compression ratios, yet they have the advantage of being hardware friendly: if a neuron is removed from a layer, it simply reduces the dimension of the weight matrix, which can be implemented natively. The current body of the pruning methods perform both structured and unstructured pruning and can be divided into two groups: the heuristic methods that propose some ranking of the weights (or neurons), known as *saliency ranking*, and the methods that achieve pruned models by formulating an optimization problem involving *sparsifying penalties*.

2.2.1 Saliency ranking methods

The methods in this category rank weights (or neurons) of the pre-trained neural network according to some saliency measure, prune certain fraction of it, and optionally retrain the remaining weights on the original model task. The main idea of the saliency methods is to estimate (often heuristically) what happens to the model loss $L(\mathbf{w})$ if a single weight w_i becomes zero. At its best, such an approximation is only accurate for a single weight removal, as it cannot account for the combined effects of multiple weights being simultaneously pruned. Yet, saliency methods are often used as robust baselines due to their simplicity. We review several methods revolving around this idea next.

Assume we have a pre-trained neural network, with weights $\mathbf{w}$ minimizing a certain loss function L, i.e. $\mathbf{w} = \arg\min_{\mathbf{w}} L(\mathbf{w})$. The pruning of the weight w_i can be written as a perturbation $\mathbf{w} - w_i\mathbf{e}_i$, where $\mathbf{e}_i$ is a unit vector along i-th axis. The change of the loss under such perturbation is given by Taylor's expansion:

$$L(\mathbf{w} - w_i\mathbf{e}_i) \approx L(\mathbf{w}) - w_i \nabla L(\mathbf{w})^T \mathbf{e}_i + \frac{1}{2} w_i^2 \mathbf{e}_i^T \mathbf{H} \mathbf{e}_i \tag{2.3}$$

Here $\mathbf{H}$ is the matrix of the second derivatives of the loss L evaluated at $\mathbf{w}$, the Hessian matrix. Since $\mathbf{w}$ is the minimizer of the loss function (by our pre-training assumption), the $\nabla L(\mathbf{w}) = 0$. Thus, the change in the loss when w_i gets pruned is given by the third term of (2.3), which gives us the salience ranking for every weight w_i. To compute the salience, LeCun et al. [78] used a diagonal approximation to the Hessian, in which case the change of the loss is given by

$$S_i = L(\mathbf{w} - w_i\mathbf{e}_i) - L(\mathbf{w}) = \frac{1}{2} w_i^2 \mathbf{H}_{ii},$$

where $\mathbf{H}_{ii}$ is the i-th diagonal item of $\mathbf{H}$. Hassibi and Stork [42] considered a more general formulation of (2.3) where during the pruning of w_i we are allowed to modify other weights by $\delta\mathbf{w}$-perturbation such that $\mathbf{w}_i + \delta w_i = 0$. In such case, the subsequent change in the value of the loss is given by saliency of

$$S_i = \frac{1}{2} \frac{w_i^2}{(\mathbf{H}^{-1})_{ii}}.$$

This approach requires the computation of the inverse of the Hessian. Instead of using the full Hessian, Hassibi and Stork [42] propose to use the Gauss-Newton approximation for the Hessian (involving only Jacobian terms). The method of Hassibi and Stork [42] is a generalization of the method of LeCun et al. [78] and empirically found to be working better. Unfortunately, computation of the full Hessian matrix does not scale well with the current sizes of deep neural networks, as it requires $O(N^2)$ storage for a network with N parameters. Even diagonal approximation of LeCun et al. [78] is hard to compute due to the complexity of backpropagation of the second-order derivatives. Therefore, numerous follow-up works were proposed to rectify the challenges with Hessian based methods. One approach of handling the size of the Hessian matrices is by applying the method of Hassibi and Stork [42] in layer-wise fashion, e.g, as in the work of Dong et al. [29].

Hessian computation can be avoided if the first-order expansion of the equation (2.3) is to be analyzed as it was done in the works of Molchanov et al. [96], Mozer and Smolensky [97]. For this to work out, the assumption of $\nabla L(\mathbf{w}) = 0$ must be dropped, i.e., we no longer assume that pruning is applied to a fully pre-trained model minimizing the $L(\mathbf{w})$. In such case, Mozer and Smolensky [97] define the saliency of the weight to be:

$$S_i = \left| w_i \frac{\partial}{\partial w_i} L(\mathbf{w}) \right|.$$

Molchanov et al. [96] use this saliency measure to rank entire filter groups, where the saliency of the filter is given as the sum of the weight saliencies within the group.

Interestingly, as noted by Hassibi and Stork [42], the Hessian-based saliency approach motivates the *magnitude-based pruning*. If we assume diagonal isotropic Hessian in (2.3), the salience of the weight is its magnitude ($S_i = w_i$). This yields a simple yet efficient method when applied to the pruning of deep neural networks [37, 39, 132], and can be generalized to the pruning of the filters using the ℓ_1 norm [81]. However, magnitude-based pruning is a local, naive approximation of the pruning process and other approaches can achieve higher sparsities.

2.2.2 Sparsifying penalties

A more principled approach to achieve a pruned neural network is by application of the sparsifying penalty $\Omega(\mathbf{w})$ and training the following loss jointly:

$$\min_{\mathbf{w}} L(\mathbf{w}) + \lambda\,\Omega(\mathbf{w}).$$

Here, λ is user defined hyper-parameter controlling the amount of desired sparsity. The initial penalties studied in this context used modifications of weight decay, which were influenced by Tikhonov's regularization of statistical models. Particularly, the ℓ_2 penalty of $\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2$ penalizes the magnitudes of the weights, and forces all of the weights to be smaller. Such penalty can be easily handled by the gradient descent methods, and have been used throughout in the literature. However, the ℓ_2 penalty does not have a strong sparsifying effect. To remedy it, as discussed in works of Weigend et al. [118] and Hanson and Pratt [40], Rumelhart proposed another weight decay of the form of

$$\Omega(\mathbf{w}) = \sum_i \frac{w_i^2}{1 + w_i^2}.$$

This penalty will not affect weights with large magnitudes, as $w_i^2 \approx w_i^2 + 1$ when w_i is large, however will penalize smaller weights driving them to zero.

Another type of penalty, the ℓ_1-norm of the form $\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$, has gained significant attention in the convex optimization literature due to convexity of such penalty and strong sparsifying properties. In fact, the linear regression model with ℓ_1 penalty is well known as a LASSO problem, and has efficient learning algorithms [43, ch. 5]. While the ℓ_1 norm penalizes only individual weights, it can be modified into a *group* penalty to target the pruning of entire neurons of the network. One example of such group penalty is $\ell_{2,1}$ proposed by Ding et al. [27]. The approaches involving ℓ_1 penalty on the weights and groups has been widely used for neural network compression. In the works of Liu et al. [90] and Ye et al. [128] the ℓ_1 penalty is applied to batch-normalization scalars, essentially pruning the entire neuron groups. In the works of Alvarez and Salzmann [3], Lebedev and Lempitsky [76], Wen et al. [119] the group $\ell_{2,1}$ penalty is used. Other sparsifying penalties based on ℓ_1 norm and its variants are being investigated as well [127].

Learning in the presence of ℓ_1 Stochastic gradient descent (SGD), a standard tool for training of deep nets, has several challenges when training ℓ_1-norm penalized models due to: a) the ℓ_1 norm is not differentiable at zero which makes it hard to state convergence guarantees (if any) when approaching pruned solutions and b) stochasticity of SGD introduces the noise for every trained weight w_i, making it impossible to identify whether w_i is actually pruned or not. Some approaches, ignore these challenges and proceed with SGD, performing a final heuristic thresholding of the weights at the end of the training [76, 119]. Other works adopt the algorithms like ISTA and FISTA [8], which are guaranteed to converge in convex case, however, it is unclear how do such convergence guarantees translate for modern neural networks with nonconvex loss surfaces.

2.3 Quantization

The neural network quantization is the process of forcing the weights of the neural network to be shared, that is every weight w_i must come from a codebook of K entries $\mathcal{C} = \{c_1, c_2, \ldots, c_K\}$. The codebook entries themselves might be fixed by hand, for example with binary ($\mathcal{C} = \{0, 1\}$) or ternary codebooks ($\mathcal{C} = \{-1, 0, +1\}$); partially adaptive, where only a learned rescaling of the codebook is allowed; or fully adaptive without any constraints on the codebook entries. The compression is achieved via a) the savings in the storage of the net: instead of storing N floating-point weights w_i (e.g., $N \times 32$ bits), we only need to store index into a codebook plus codebook itself ($\lceil \log K \rceil \times N + K \times 32$ bits) and b) savings in inference: if both weights and activations are quantized using a suitable codebook (say integer-only) forward pass through the net can be computed more efficiently.

While neural network quantization was studied as early as in 90s [32, 114], the interest in this compression mechanism was revitalized with recent push into neural network compression. The current landscape of quantization work can largely be divided into:

- **Quantization aware training**, QAT. In QAT setting we assume that we have full control over the neural network weights and the training data, so we can execute a complicated optimization pipeline [5, 6, 16, 49, 65].

- **Post training quantization**, PTQ. In PQT setting we assume that we only have access to the trained network that needs to be quantized, without the accompanying dataset that the network was trained on. It is expected that the entire quantization pipeline can be executed quickly, typically within minutes [7, 74, 83, 98, 121].

In terms of error-compression tradeoff of the resulting quantized networks, the quantization-aware training is the leading approach. A general setting of such methods is to solve a problem of the form:

$$\min_{\mathbf{w},\mathcal{C}} L(\mathbf{w}) \quad \text{s.t.} \quad w_i \in \mathcal{C}, \quad \forall i = 1, \ldots, N. \tag{2.4}$$

In terms of the solutions of this problem we can outline three big categories: a) methods that use modification of backpropagation b) methods that use constrained optimization machinery. We discuss these methods next.

Methods that modify backpropagation A large number of works [5, 6, 22, 23, 49, 65, 74] uses an ad-hoc solution to the problem (2.4) that involves modification of the stochastic gradient descent. While exact details differ across the methods, the central idea is to maintain two copies of the weights: full precision ones and their quantized copy. During the forward pass through the network the quantized weights are used. Although the gradient wrt quantized weights is undefined due to non-differentiability of quantization constraints, in these methods, the gradient is computed as if no quantization constraints existed. This fake gradient is then used to update *full-precision* weights, and then a new quantized copy is computed from newly updated full weights.

Methods that use constrained optimization machinery A more principled way to solve problem (2.4) is to rely on machinery of constrained optimization. Such methods involve formulating the penalized version of the problem and solving it using alternating optimization [16, 79] or rely on projected gradient descent [130, 131].

2.3.1 Scalar quantization using squared distortion

When quantizing the weights of the neural networks, many algorithms often rely on solving squared distortion quantization problem defined with respect to the weights $\mathbf{w} = \{w_1, \ldots, w_N\}$ as :

$$\begin{aligned} \min_{\mathbf{Z}, \mathcal{C}} \quad & \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} (w_i - c_k)^2 \\ \text{s.t.} \quad & \mathbf{z}_i^T \mathbf{1} = 1, \quad \mathbf{z}_i \in \{0,1\}^K. \end{aligned} \tag{2.5}$$

Here, $\mathbf{Z}$ is a matrix containing the binary assignment vectors: each weight w_i must be assigned to a single codebook c_k with $z_{ik} = 1$. This problem might be familiar to many readers: it is a 1d version of the k-means clustering problem [93]. Although, for dimensions $d \geq 2$ and number of clusters $k \geq 2$ this problem is NP-hard [2, 24, 94], this is not the case for the scalar version. In fact, Bruce [11] gave a $O(NK^2)$ solution using dynamic programming (DP). Wu and Rokne [124] improved Bruce's DP algorithm to have a runtime of $O(NK \log K)$ using divide-and-conquer approach, and Wu [123] further reduced the runtime to $O(NK)$ relying on matrix searching techniques.

The continuous version of problem (2.5), assuming a data distribution $p(w)$ on weights and modifying sum to expectation, was studied by Lloyd [91] in the context of pulse-code modulation. Lloyd gave the closed-form solutions for quantizers of Gaussian and Laplacian distributions and introduced an alternating optimization algorithm, which is a 1d version of the k-means algorithm.

Special forms

Many special forms of (2.5) appear as substeps in network quantization as well. One particular case is when codebook entries are fixed but are allowed to be rescaled by single α:

$$\begin{aligned} \min_{\alpha, \mathbf{Z}} \quad & \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} (w_i - \alpha\, c_k)^2 \\ \text{s.t.} \quad & \mathbf{z}_i^T \mathbf{1} = 1, \quad \mathbf{z}_i \in \{0,1\}^K, \quad \forall i = 1, \ldots, N \end{aligned} \tag{2.6}$$

Despte the lack of guarantees, a popular method in solving the rescaled scalar quantization problem of (2.6) is alternating optimization akin to k-means with a step over α and a step over assignments $\mathbf{Z}$. For instance, Hwang and Sung [49] used the alternating optimization for a case of $\mathcal{C} = \{-1, 0, 1\}$, Anwar et al. [6] used it for the uniform integer codebooks, and Leng et al. [79] employed it for the powers-of-two codebooks.

Provable optimal algorithm for some specific cases of problem (2.6) and for the general formulation have been derived too. Rastegari et al. [104] gave a solution for scaled binary quantization where $\mathcal{C} = \{-1, 1\}$, Carreira-Perpiñán and Idelbayev [16] and Yin et al. [130] gave a solution for optimal scaled ternarization with $\mathcal{C} = \{-1, 0, 1\}$. For the generic case with arbitrary $\mathcal{C}$, the globally optimal algorithm running in $O(NK \log K)$ was given by Idelbayev et al. [60].

INT8 quantization The scaled INT8 case of problem (2.6), where the weights are quantized into the rescaled codebook of $\mathcal{C} = \{0, \pm 1, \pm 2, \dots, \pm 2^7\}$, has gained a significant interest.

The solutions to the INT8 version of (2.6) available in the literature can be divided into the following categories: alternating optimization solutions [5, 6, 49, 111, 134], heuristics based on maximum values [65, 121] or percentiles [121], grid search search techniques [6, 21, 49, 89], and analytical solutions assuming a certain distribution on datapoints [7, 13, 19, 30, 83]. None of these approaches, except for the finely-spaced grid search, can guarantee a global optimum of INT8 quantization problem on arbitrary data. However, an exhaustive sweep through the entire search space for α-values is expensive; thus, some approximations are used: Hwang and Sung [49] first find locally optimal solution using alternating optimization and then improve it by a limited grid search; Choukroun et al. [21] fix the number of points in the grid; Liu et al. [89] give a heuristic rule on how finely to space the grid.

Only recently, Idelbayev et al. [60] gave the globally optimal algorithm that can solve problem (2.6) for any codebook $\mathcal{C}$ including the INT8 version.

2.4 Model compression software

The field of model compression has grown enormously in the recent years resulting in plethora of algorithmic approaches, research projects and software. At present, many ad-hoc solutions have been proposed that typically solve *only one specific type of compression*: quantization [16, 104, 138], pruning [37, 88, 119], low-rank decomposition [25, 26, 66, 80, 109, 113, 120, 125, 126, 135] or tensor factorizations [26, 77], and others. In this section we limit our attention to the software aspect of the neural network compression and overview the supported compression schemes among the software, available codes, and recently proposed compression frameworks.

Individual compressions The majority of neural network compression research is available as individual projects and recipes tailored for a particular compression and model. Usually it is released as a companion code for published research paper, e.g. see [113, 116, 125]. Some repositories combine several compression recipes in a single place: e.g., Tensorpack[1] or the fork of the Caffe library by Wei Wen[2].

Out of many individual compressions proposed in the literature, the quantization aware training of Jacob et al. [65] has gained popularity and became a standard feature of major deep-learning frameworks. TensorFlow, Pytorch, MxNet and others independent projects like ADaPTION [95], Mayo [137], FINN-R[10] and TensorQuant [92] natively support both training of such quantized models and allow an efficient inference afterwards.

Efficient inference frameworks Relatively mature software is available if the goal is not to compress the model (by changing the weights accordingly), but to run the model as efficiently as possible on a given hardware. Many frameworks target the mobile deployment regime and allow to convert (compile) already trained neural network to utilize the hardware-enabled fast computations: for instance, through usage of edge TPU-s on Pixel 4 (Pixel Neural Core) or Neural Engine

[1] `https://github.com/tensorpack/tensorpack/tree/master/examples`

[2] `https://github.com/wenwei202/caffe`

on iPhones. Examples of such frameworks include Tensorflow Lite[3], PyTorch Mobile[4], Apple Core ML[5], Nvidia's TensorRT[6], Qualcomm's Neural Processing SDK[7], Xlinix's FINN[8], Facebook's QNNPack[9], and many others.

A generalization of this concept is to efficiently deploy and compile the dataflow of the inference/backward pass for an arbitrary set of hardware. Some examples include packages like Facebook's Glow[10], Google's XLA[11], or Apache TVM[12].

Compression frameworks The diversity of compression mechanisms and limited support by deep learning frameworks led to the development of specialized software libraries such as Distiller [141], NCCF [73], and PocketFlow [122]. These frameworks gather multiple compression schemes and corresponding training algorithms into a single framework, and make it easier to apply the compressions to new models. Some of these frameworks allow to apply multiple compression simultaneously to disjoint parts of a single model, however most of the supported schemes can be applied with per-model granularity only. Additionally, the underlying compression algorithms do not share the same algorithmic base thus requiring a substantial understanding of many hyper-parameters for every compression-algorithm pair to efficiently tune the settings.

[3] `https://www.tensorflow.org/lite`

[4] `https://pytorch.org/mobile/home/`

[5] `https://developer.apple.com/documentation/coreml`

[6] `https://developer.nvidia.com/tensorrt`

[7] `https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk`

[8] `https://xilinx.github.io/finn/`

[9] `https://engineering.fb.com/ml-applications/qnnpack/`

[10] `https://ai.facebook.com/tools/glow`

[11] `https://tensorflow.google.cn/xla?hl=en`

[12] `https://tvm.apache.org/`

Chapter 3

Overview of the Learning-Compression algorithm

In this chapter, we give an overview of the Learning Compression (LC) algorithm used in our research. For the full details on the theoretical framework we refer the reader to Carreira-Perpiñán [15].

Assume we have a previously trained model with weights $\mathbf{w}$, which were obtained by minimizing some loss function $L(\mathbf{w})$. This is our *reference* model, which represents the best loss we can achieve without compression. Here we omitted the exact definition of the weights $\mathbf{w}$, but for now, let us assume it has P parameters. In the learning-compression framework, the compression is defined as finding a low-dimensional parameterization $\boldsymbol{\Delta}(\boldsymbol{\Theta})$ of the weights $\mathbf{w}$ in terms of Q-sized parameter $\boldsymbol{\Theta}$, with $Q < P$.

In the framework, the compression and decompression are regarded as mappings, while in the signal processing literature they are usually seen as algorithms, for example, lossless compression algorithm of Ziv and Lempel [140]. Formally, the *decompression mapping* $\boldsymbol{\Delta}$ maps a low-dimensional parameters $\boldsymbol{\Theta}$ to the uncompressed model weights $\mathbf{w}$:

$$\boldsymbol{\Delta}\colon\ \boldsymbol{\Theta} \in \mathbb{R}^Q \rightarrow \mathbf{w} \in \mathbb{R}^P,$$

and the *compression mapping* behaves as its "inverse":

$$\boldsymbol{\Pi}(\mathbf{w}) = \arg\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2.$$

The goal of model compression is to find such $\mathbf{\Theta}$ that its corresponding decompressed model has (locally) optimal loss for a cost of interest. Therefore the *model compression as a constrained optimization* problem is defined as:

$$\min_{\mathbf{w},\mathbf{\Theta}} L(\mathbf{w}) + \lambda\, C(\mathbf{\Theta}) \quad \text{s.t.} \quad \mathbf{w} = \mathbf{\Delta}(\mathbf{\Theta}). \tag{3.1}$$

Here, the term $\lambda\, C(\mathbf{\Theta})$ with $\lambda > 0$ is intended to represent the cost of the deployed compressed model in terms of quantities of interest: energy, size, compute, etc. The problem in eq. (3.1) is constrained, nonlinear, and potentially non-differentiable wrt $\mathbf{\Theta}$ (e.g., when compression is binarization). To efficiently solve it we convert this problem to an equivalent formulation using penalty methods, for which we can either use quadratic penalty (QP) or augmented Lagrangian (AL):

$$\mathcal{L}_{\text{QP}}(\mathbf{w}, \mathbf{\Theta}; \mu) = L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2 + \lambda\, C(\mathbf{\Theta}) \tag{3.2}$$

$$\mathcal{L}_{\text{AL}}(\mathbf{w}, \mathbf{\Theta}, \boldsymbol{\beta}; \mu) = L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2 + \boldsymbol{\beta}^T(\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})) + \lambda\, C(\mathbf{\Theta}). \tag{3.3}$$

Under standard assumptions (differentiable L, C, and $\mathbf{\Delta}$), the stationary points at $\mu \to \infty$ of eq. (3.2) and eq. (3.3) coincide with the stationary point of constrained optimization problem (3.1). We will be using the QP formulation of eq. (3.2) throughout this paper to make derivations easier, though, in practice we implement the AL version, eq. (3.3), which has an additional vector $\boldsymbol{\beta}$ of Lagrange multipliers. The QP version can be obtained from the AL version by setting $\boldsymbol{\beta} = 0$ and skipping the multipliers update step.

To obtain the LC algorithm we apply an alternating optimization to eq. (3.2) wrt model parameters $\mathbf{w}$ and compression parameters $\mathbf{\Theta}$. This results into an algorithm that alternates two generic steps while slowly driving the penalty parameter $\mu \to \infty$:

- **L (learning) step:** $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2$. This is a regular training of the uncompressed model but with a quadratic regularization term. *This step is independent from the compression.*

- **C (compression) step:** $\min_{\mathbf{\Theta}} \|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2 + \lambda\, C(\mathbf{\Theta})$. When $\lambda = 0$ this means finding the best (lossy) compression of $\mathbf{w}$ (the current uncompressed

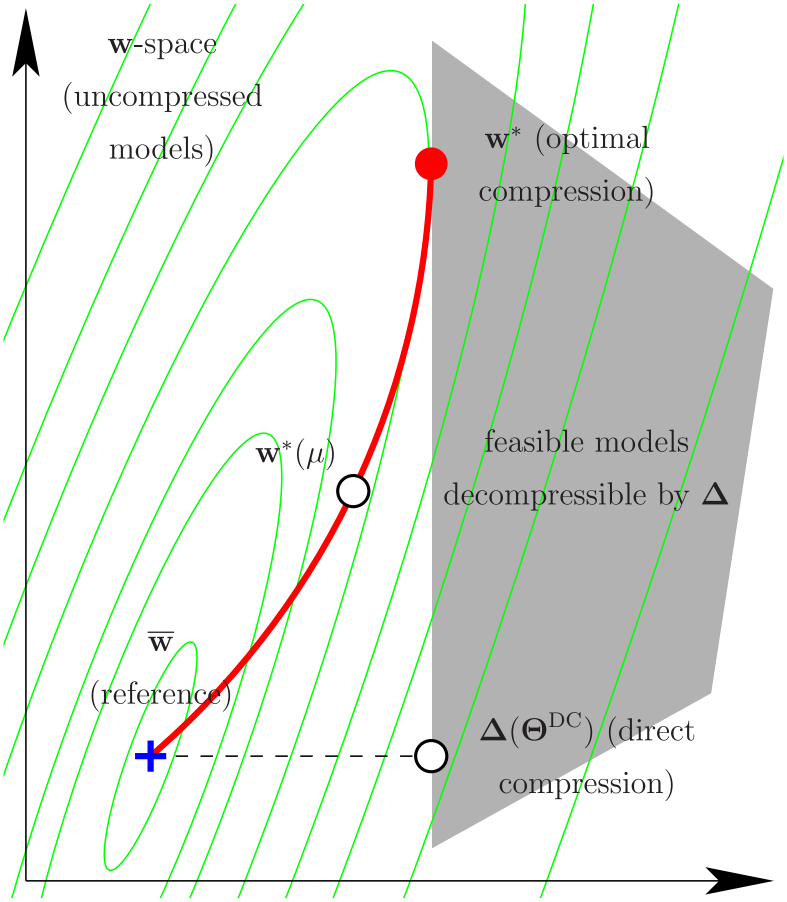


Figure 3.1: The illustration of the model compression definition given by problem (3.1) for $\lambda = 0$. The loss function $L(\mathbf{w})$ is defined over the entire $\mathbf{w}$ space, depicted with green contours, and has a minimum at point $\overline{\mathbf{w}}$. The space of decompressible models (given by the form of of $\mathbf{\Delta}$) is illustrated in gray. Directly compressing the pre-trained model by setting $\mathbf{\Theta}^{\mathrm{DC}} = \mathbf{\Pi}(\overline{\mathbf{w}})$ results in sub-optimal solution. To obtain the constrained minima of the problem (the point $\mathbf{w}^*$), the LC algorithm alternates between L and C steps while driving parameter $\mu \to \infty$, which follows the path $\mathbf{w}^*(\mu)$. The figure is obtained from Carreira-Perpiñán [15].

model) in the ℓ_2 sense (orthogonal projection on the feasible set), and acts as the inverse of mapping $\mathbf{\Delta}$. For a nonzero λ, this step's solution finds such a $\mathbf{\Theta}$ that is close to $\mathbf{w}$, but also respects our compression cost C. Notably, *this step is independent from the model loss*, and thus independent of the dataset.

Figure 3.1 illustrates the idea of model compression as constrained optimization, and depicts the traced solution $\mathbf{w}^*(\mu)$ during the optimization, and in Fig-

```
input training data and model with parameters $\mathbf{w}$
$\mathbf{w} \leftarrow \overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$    pre-trained model
$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta}^{\mathrm{DC}} = \boldsymbol{\Pi}(\overline{\mathbf{w}})$    init compression
$\boldsymbol{\beta} \leftarrow \mathbf{0}$
for $\mu = \mu_0 < \mu_1 < \dots < \infty$
    $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta}) - \frac{1}{\mu}\boldsymbol{\beta}\|^2$    L step
    $\boldsymbol{\Theta} \leftarrow \arg\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \frac{1}{\mu}\boldsymbol{\beta} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2 + \lambda\, C(\boldsymbol{\Theta})$    C step
    $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \mu(\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta}))$    multipliers step
    if $\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|$ is small enough then exit the loop
return $\mathbf{w}$, $\boldsymbol{\Theta}$
```

Figure 3.2: The pseudocode of the learning-compression (LC) algorithm using augmented Lagrangian formulation. Setting the $\boldsymbol{\beta} = 0$ and skipping the multipliers step will recover quadratic penalty (QP) formulation of the LC algorithm.

ure 3.2 we give the algorithm's pseudocode using augmented Lagrangian. The LC algorithm defines a continuous path $(\mathbf{w}(\mu), \boldsymbol{\Theta}(\mu))$ indexed by μ. The beginning of this path, at $\mu = 0^+$, corresponds to training the reference model and then compressing it disregarding the loss (*direct compression*), a simple but suboptimal approach popular in practice.

Optimization of the L and C steps The L step is a minimization problem over the weights $\mathbf{w}$, and can be solved using any optimization method. For neural networks we typically use stochastic gradient descent (SGD). Since compression parameter $\boldsymbol{\Theta}$ enters this problem as a constant regardless of chosen compression type, all L steps for any combination of compressions will have exactly the same form. The solution of the C step is specific to the desired compression types and might take various forms. However, since the C-step problem is disentangled from the model and the dataset, and has a form of ℓ_2 minimization, solving it is a much easier problem. In fact, as we have discovered in the course of our research, for certain compression choices the C-step problem is well studied and has a history of its usage on its own merit in fields of data and signal compression.

To summarize, our approach is based on solid optimization principles, with guarantees of convergence under standard assumptions. It formulates the problem of model compression in a way that is intuitive and amenable to efficient optimization. The form of the actual algorithm is obtained systematically by judiciously applying mathematical transformations to the objective function and constraints. For example, if one wants to optimize the cross-entropy over a certain type of neural net, and represent its weights via a quantized codebook, then the L and C steps necessarily take a specific form. If one wants instead to represent the weights via low-rank matrices, a different C step results, and so on. The resulting algorithm is not based on combining backpropagation training with heuristics, such as pruning weights on the fly, which may result in suboptimal results or even non-convergence.

Chapter 4

Low-rank compression with automatic rank selection

A fundamental, yet, often not recognized problem in low-rank compression of the neural networks is the problem of rank selection. This is an understandable oversight because rank selection is an easy process for certain well known cases. Say we want to find a low-rank decomposition of matrix $\mathbf{W}$ with a certain approximation error. The solution of this problem is computationally convenient: obtain a singular value decomposition (SVD) of the matrix $\mathbf{W}$ as product of $\mathbf{USV}^T$ (where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{S}$ is diagonal matrix containing singular values in sorted order), and pick as many singular values as necessary until the desired approximation error is achieved. Importantly, we do not need to solve a new optimization problem for each target approximation error: we get all solutions at once due to the special structure of the problem, thus a single SVD is sufficient.

Similarly, this property hold in a special case of model compression problem of obtaining a low-rank linear regression fit. This problem is known as reduced rank regression (RRR) in statistics [64, 106] and can be solved for all target ranks using a single SVD of the specially formed data matrix.

Let us now take a look why rank selection is a much harder problem in the case of a deep neural network. For a deep net we want to find both the rank and the weights (matrix coefficients) for each layer so that some desired compression cost is minimized and the ranks are constrained. The solution is not given anymore by computing a single SVD and examining the singular values. The problem simplifies if we know the ranks beforehand: in such case we can directly optimize the low-rank weights by rewriting them as product of low-rank matrices ($\mathbf{U}\mathbf{V}^T$) and then fall back to standard deep learning tools: i.e., use automatic differentiation coupled with stochastic gradient descent (SGD). In fact, a large body of low-rank compression literature (see section 2.1) follow this line of work and rely on heuristics to chose "good" rank configurations, however, the resulting models are by no means optimal: simply because the ranks themselves were not part of the optimization.

This makes it obvious that the real problem is in determining optimal values for the set of ranks $r_1, \dots, r_K$. It also shows that the problem can be seen as a special case of architecture optimization, where we search both over architectures (i.e., the number of hidden units, or rank, within each layer) and over values of the matrices' weights. Hence, this is a hard, combinatorial problem which is exponential on the number of layers. Specifically, in a net with K layers of weights each having a maximum rank of R there are R^K combinations of rank choices, and R can be thousands in large nets.

In this chapter, we use constrained optimization formulation for the rank selection problem and derive a good, approximate, and efficient solution to it. Our proposed formulation will jointly learn ranks and weights of the deep neural networks by exploring different rank configuration on the fly, during the optimization. In practice, this does not result in much longer training time in comparison to training of the reference network in the first place; and, as we experimentally validate, yields models that are comparable or better when comparison to low-rank methods in the literature and other compression techniques.

4.1 Problem formulation

Let us assume we have a reference model with K layers and the weights $\mathbf{w} = \{\mathbf{W}_1, \dots, \mathbf{W}_K\}$, where $\mathbf{W}_k$ is the weight matrix of the layer k with shape of $a_k \times b_k$. For simplicity, we derive the algorithm assuming fully connected layers, yet the application to other (say, convolutional) layers is straightforward. We want to determine the weights and the ranks of every layer in a such way that the resulting low-rank model performs as good as possible on its task loss L, while simultaneously having the lowest compression cost C among the family of low-rank models. Thus, we want to solve the following optimization problem:

$$\min_{\mathbf{w}} L(\mathbf{w}) + \lambda\, C(\mathbf{w}) \quad \text{s.t.} \quad \operatorname{rank}(\mathbf{W}_k) = r_k \leq R_k, \quad \forall\, k = 1, \dots, K. \tag{4.1}$$

Here, R_k is the maximum possible rank for matrix $\mathbf{W}_k$, i.e. $R_k \leq \min(a_k, b_k)$ and λ is a user defined hyperparameter which trades off model accuracy (the loss L) to compression cost C. This formulation penalizes the models that have high cost C, thus performing *a model selection*. The cost function C measures the quantity of interest we would like to compress (say, storage space in bits), and we explicitly define C as a function of the layers' ranks, for which we choose it to be separable over the layers in the following way:

$$C(\mathbf{w}) = C(\mathbf{r} = \{r_1, \dots, r_K\}) = C(r_1) + C(r_2) + \cdots + C(r_K). \tag{4.2}$$

This cost function is generic enough to handle following quantities of interest:

- **Storage**. The r-rank weights $\mathbf{W}_k$ can be decomposed as a product $\mathbf{U}_k\mathbf{V}_k$, and require the storage of matrix $\mathbf{U}_k$ and $\mathbf{V}_k$ which have $a_k \times r$ and $r \times b_k$ entries respectively. Thus, storage penalty will have the form of $C(r_k) = r_k \times (a_k + b_k)$

- **FLOPs**. For the fully connected layers, the forward pass through a layer requires computation of the product of $\mathbf{W}_k\mathbf{x}$ where $\mathbf{x}$ is an input to the layer. If the matrix has low rank r, the product can be efficiently computed by $\mathbf{U}_k\,(\mathbf{V}_k\mathbf{x})$, where inner parenthesis is evaluated first. Such procedure requires $r \times (a_k + b_k)$ floating point additions and multiplications (FLOPs). For convolutional layers, each weight matrix is applied M times, thus a generic FLOPs penalty has a form of $C(r_k) = M \times r_k \times (a_k + b_k)$.

4.2 Optimization algorithm

The optimization problem of eq. (4.1) is mixed-integer programming involving optimization over weights and implicitly over ranks, which is hard to optimize. To make it amenable to efficient optimization, we proceed by introducing the auxiliary variables and convert the problem into *"model compression as constrained optimization"* formulation of eq. (3.1), and then derive the LC algorithm. Let us introduce a matrix $\mathbf{\Theta}_k$ for every layer $k = 1, \dots, K$ with constraints $\mathbf{W}_k = \mathbf{\Theta}_k$:

$$\begin{aligned} \min_{\mathbf{W}, \mathbf{\Theta}, \mathbf{r}} \quad & L(\mathbf{W}) + \lambda\, C(\mathbf{r}) \\ \text{s.t.} \quad & \mathbf{W}_k = \mathbf{\Theta}_k,\ \operatorname{rank}(\mathbf{\Theta}_k) = r_k \leq R_k,\ k = 1, \dots, K \end{aligned} \tag{4.3}$$

We then apply the quadratic penalty on equality constraints:

$$\begin{gathered} Q(\mathbf{W}, \mathbf{\Theta}, \mathbf{r}; \mu) = L(\mathbf{W}) + \lambda\, C(\mathbf{r}) + \frac{\mu}{2} \sum_{k=1}^{K} \left\| \mathbf{W}_k - \mathbf{\Theta}_k \right\|^2 \\ \text{s.t.} \quad \operatorname{rank}(\mathbf{\Theta}_k) = r_k \leq R_k,\ k = 1, \dots, K \end{gathered}$$

and optimize it while driving $\mu \to \infty$ by alternating over weights $\mathbf{w}$ and auxiliary variables $\boldsymbol{\theta} = \{\mathbf{\Theta}_1, \dots, \mathbf{\Theta}_K\}$. This will result in learning compression algorithm with following L and C steps:

- **L step:** $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2} \sum_{k=1}^{K} \left\| \mathbf{W}_k - \mathbf{\Theta}_k \right\|^2$ has the form of standard neural network training with ℓ_2 weight decay, for which we will be using stochastic gradient descent (SGD).

- **C step:** $\min_{\boldsymbol{\theta}, \mathbf{r}} \lambda\, C(\mathbf{r}) + \frac{\mu}{2} \sum_{k=1}^{K} \left\| \mathbf{W}_k - \mathbf{\Theta}_k \right\|^2$ s.t. $\operatorname{rank}(\mathbf{\Theta}_k) = r_k \leq R_k$, for $k = 1, \dots, K$. This is a joint rank selection and weight fitting problem for which we provide an efficient solution involving singular value decomposition (SVD).

C step solution Since the cost function C separates over layers, see eq. (4.2), the entire C-step problem breaks down into K subproblems of:

$$\min_{\boldsymbol{\theta}_k, r_k} \quad \lambda\, C(r_k) + \frac{\mu}{2} \left\| \mathbf{W}_k - \mathbf{\Theta}_k \right\|^2 \quad \text{s.t.} \quad \operatorname{rank}(\mathbf{\Theta}_k) = r_k \leq R_k. \tag{4.4}$$

If the optimal rank r_k^* was known apriori, then we only need to find

$$\min_{\mathbf{W}_k} \quad \frac{\mu}{2}\|\mathbf{W}_k - \boldsymbol{\Theta}_k\|^2$$

as the $C(r_k)$ is constant. In such case, we recognize this problem as finding best r_k^*-rank approximation of $\mathbf{W}_k$, and the corresponding optimal $\boldsymbol{\Theta}_k^*$ is given by the Eckhart-Young theorem [35, th. 2.4.8]. Assuming w.l.o.g. $a_k \geq b_k$ and let $\mathbf{W}_k = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$ be the SVD of $\mathbf{W}_k$, where $\mathbf{U}_k$ of $a_k \times b_k$ and $\mathbf{V}_k$ of $b_k \times b_k$ are orthogonal matrices, and $\mathbf{S}_k = \operatorname{diag}(s_1, \dots, s_{b_k})$ with $s_1 \geq \cdots \geq s_{b_k} \geq 0$ (sorted singular values). Then the optimal $\boldsymbol{\Theta}_k^*$ corresponding to r_k^* is given as:

$$\boldsymbol{\Theta}_k^* = \mathbf{U}_k(:, 1\!:r_k^*)\, \mathbf{S}_k(1\!:r_k^*, 1\!:r_k^*)\, \mathbf{V}_k(:, 1\!:r_k^*)^T. \tag{4.5}$$

Since we do not know the r_k^* which minimizes the eq. (4.4) we simply enumerate over every possible $r_k = 1, \dots, R_k$ and compute the corresponding $\boldsymbol{\Theta}_k$ using eq. (4.5). The pair of $(r_k, \boldsymbol{\Theta}_k)$ corresponding to the minimum value of the eq. (4.4) is the global solution of the C-step problem. This enumeration requires only a single full SVD of $\mathbf{W}_k$.

Overall, the LC algorithm follows the pseudocode on Figure 3.2, and operates by training the regularized model for a while with SGD over the full-rank matrices $\mathbf{W}_1, \dots, \mathbf{W}_K$ (with a regularization term given by each low-rank matrix $\boldsymbol{\Theta}_k$), the L step, and then obtaining each low-rank matrix $\boldsymbol{\Theta}_k$ with currently optimal rank r_k via a SVD of $\mathbf{W}_k$, the C step. We maintain two copies of each layer's matrix: $\mathbf{W}_k$ of full rank, and $\boldsymbol{\Theta}_k$ of rank r_k determined within each C step. These copies will coincide in the limit $\mu \to \infty$ with $\mathbf{W}_k = \boldsymbol{\Theta}_k$. The automatic rank selection happens within the C step, effectively by doing a model selection over the rank of each matrix. Practically, rather than continuing to iterate L and C steps until convergence, at some iteration we fix the ranks, thereby fixing the architecture, and optimize it directly via SGD with the chain rule, which is faster.

4.3 Experimental setup

We evaluate our algorithm on multiple datasets and networks: ResNets, VGG16, and NiN on CIFAR10; AlexNet on ImageNet; and compare our results to baselines, and other relevant works. We choose both lean (ResNets) and large (AlexNet, VGG16) networks to demonstrate the power of rank-selection approach. Experiments are initialized from reasonably well-trained reference models with same or exceeding test accuracies reported in literature.

The hyper parameters of our experiments are as follows throughout all experiments with minor changes, see Idelbayev and Carreira-Perpiñán [53] for full details. *To optimize L-step* we use Nesterov's accelerated gradient method [100] with momentum of 0.9 on minibatches of size 128 (256 for MNIST), with decayed learning rate schedule of $\eta_0 \times a^m$ at m-th epoch. The initial learning rate η_0 is one of $\{0.0007, 0.001\}$, and learning rate decay is one of $\{0.98, 0.99\}$. Each L-step is run for 15 epochs (30 for MNIST). Our LC algorithm runs for j steps where $j \leq 60$, and has $\mu_j = \mu_0 \times b^j$, and we choose μ_0 to be one of $\{5 \cdot 10^{-4}, 10^{-3}\}$, and $b \in \{1.2, 1.25\}$. The C step is performed by SVD followed by rank selection. For the cost function C we use the number of floating point operations in millions, MFLOPs, which is a function of layer's rank.

We report train loss, test error, reduction ratio of storage (ρ_{storage}) and the number of floating point operations (ρ_{FLOPS}). We calculate FLOPs based on the assumption of fused multiplication and additions, treating it as one FLOP, see appendix A for exact details.

In our experiments, we use single low-rank scheme throughout a network. Experiments on the MNIST and CIFAR10 are run with scheme 1, and for the ImageNet we run experiments with both schemes.

We adopt the heuristics H1 and H2 described in section 2.1.2 as our baseline 1 and baseline 2. Both of the heuristics introduce hyper-parameter $p \in [0, 1]$ which controls the compression ratio. By changing the proportion both baselines give rank estimates which we use to decompose the original networks, and then fine-tune. We fine-tune using Nesterov SGD and set learning rates to achieve as good performance as possible. Fine-tuning happens for about twice (2×) the number of iterations required to train the reference networks, with a learning rate of 0.002 for ResNets and 0.001 for NiN and VGG-16, which decayed by 0.99 after every epoch for more details).

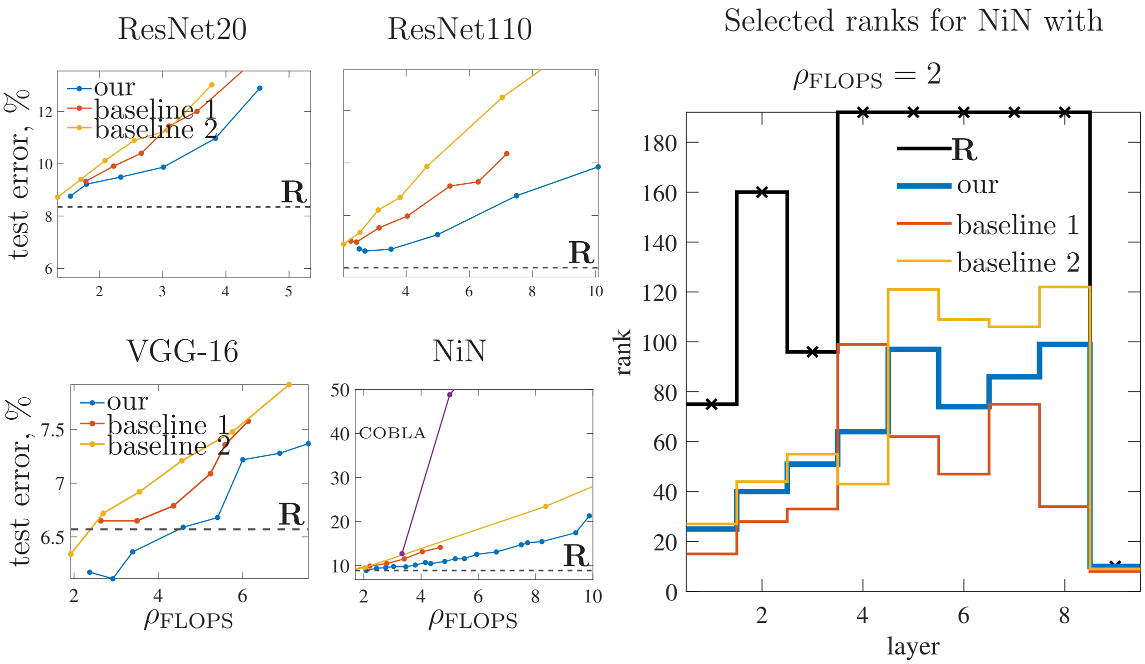


Figure 4.1: Left: Comparison of our rank selection algorithm to the baselines on CIFAR10 networks. We plot test error vs reduction ratio of FLOPs (ρ_{FLOPS}); horizontal dashed lines — reference net performances. Right: we depict selected ranks for low-rank NiNs achieving $\rho_{\text{FLOPS}} = 2$; ranks of reference NiN are given by black crossed line. The line marked as COBLA gives results of automatic rank selection method called COBLA of Li and Shi [80]

4.4 CIFAR10 experiments and comparison

We train reference ResNets of different sizes (20, 32, 56, and 110 layers) following the procedure of the original paper [45] using the code in [51]; the NiN and VGG16 (adapted for CIFAR10) are trained using the same data-augmentation as of ResNet's. We compress these networks using baselines and using our algorithm (with various values of λ), and fine-tune afterwards. For ResNets we compress convolutional layers only, as only fully-connected layer has 64×10 weights, which minimally impacts the compression.

Figure 4.1 gives a comparison with baselines on ResNets, VGG-16 and NiN trained on the CIFAR10. *Our method achieves considerably better test errors across all speed-up ratios.* This happens because the baselines are committed to the selected ranks, without the possibility to change them, whereas our algorithm explores different sets of ranks while it converges to a better one. We show the

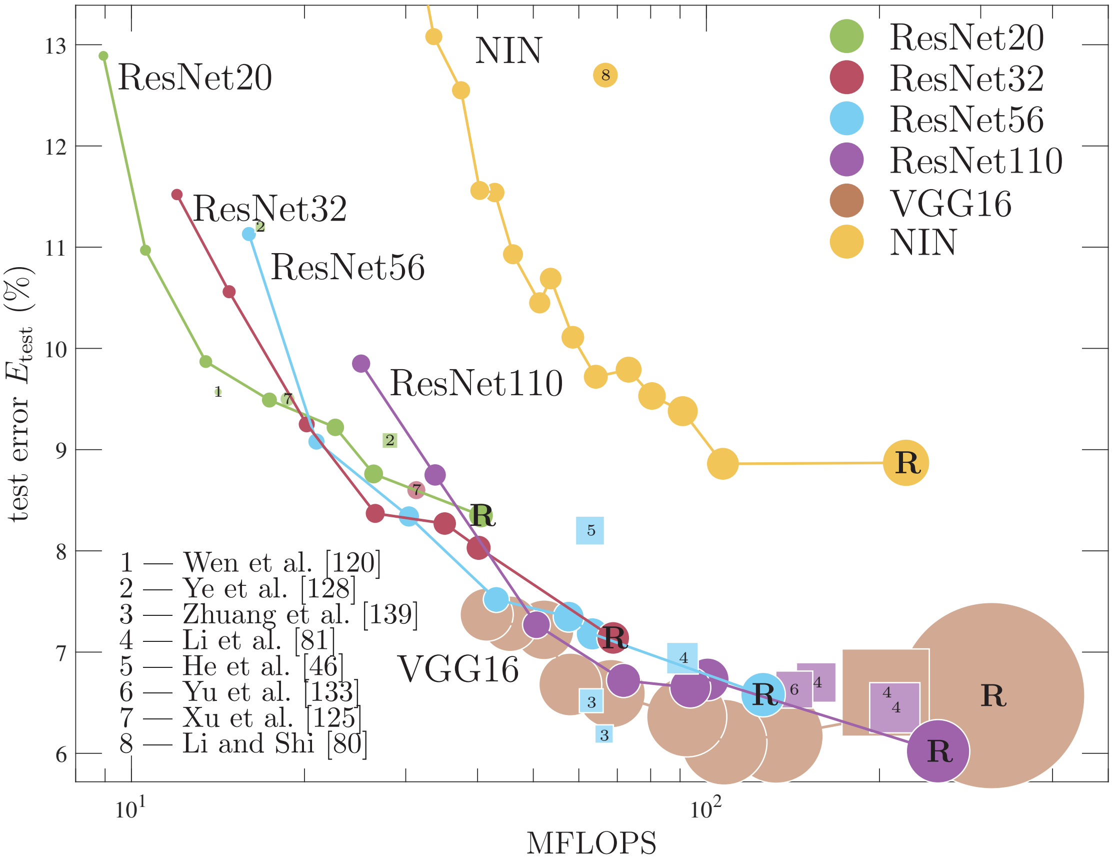


Figure 4.2: Error-compression space of test error (Y axis), inference MFLOPs (X axis) and number of parameters (ball size for each net), for ResNets, VGG16 and NIN trained on CIFAR10. Results of our algorithm over different λ values for a given network span a curve, shown as connected circles ●—●, which starts on the lower right at the reference **R** ($\lambda = 0$) and then moves left and up. Other published results using low-rank compression are shown as isolated circles labeled with a citation. Other published results involving structured filter pruning for faster inference are shown as isolated squares labeled with a citation. Each color corresponds to a different reference net. The area of a circle or square is proportional to the number of parameters in the corresponding compressed model. Ideal models are small balls (having few parameters) on the left-bottom (where both error and FLOPs are the smallest).

differences in selected ranks for the low-rank NiNs achieving 2× speed-up on the right part of Figure 4.1. While all three methods select approximately the same ranks for the first three layers, decisions for the layers 4–8 are different.

With multiple quantities of interest comparing the performance of compressed neural networks is rather tricky. The most obvious way is to report a single com-

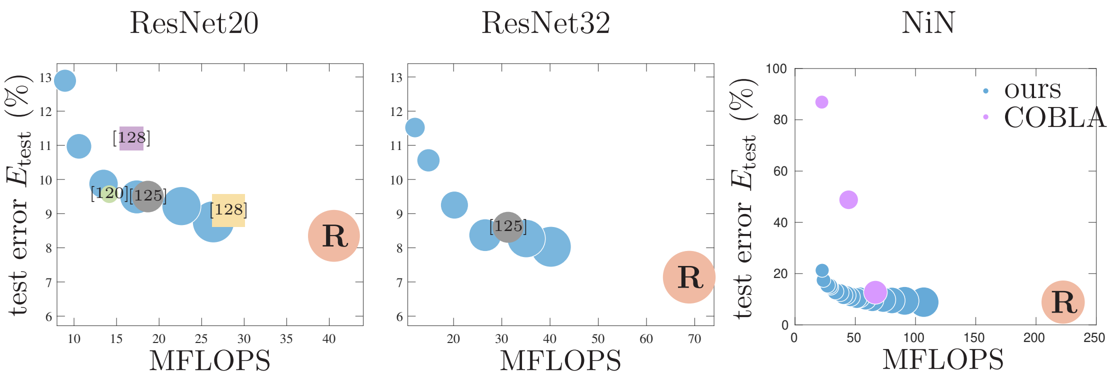


Figure 4.3: Depiction of an interplay between model FLOPs, number of parameters, resulting test error for LC Models (our), as in Figure 4.2 but separately for the ResNet20, 32. Each of the blue circles correspond to a particular compressed model via LC, it's area is proportional to number of parameters (independently normalized for each figure). Results of other compressions are given by: different colored circles — low-rank, squares — filter pruning. Right-most plot compares to joint weights and ranks learning algorithm of Li and Shi [80] called COBLA.

pression ratio in terms of the number of parameters, or speed-up. Having only one number does not necessarily reflect other important metrics, e.g., compression of parameters does not correspond to faster inference (with fewer FLOPs), and generally, not as interesting as the interplay between compressed model's performance, compression, and speed-up ratios. We should also note that compression ratios (of any kind) on its own are not representative as they can be easily inflated by compressing a larger (overparametrized) model in the first place. Therefore, to visualize and understand this tradeoff better, we decided to report achieved FLOPs, model size and test accuracy in a single plot. The Figure 4.2 depicts all our CIFAR10 results obtained via low-rank compression (as connected circles), other's results obtained via low-rank compression as labeled circles [80, 120, 125], and most importantly puts low-rank compression in perspective with other reported results for faster inference, i.e., structured filter pruning of [46, 81, 128, 133, 139], as squares (to indicate apples to oranges comparison). Ideally, we would like to have models on the left-bottom corner of this plot, where both FLOPs and error are minimal. Results trace a pareto curve, which is *mostly formed by our low-rank* compressed ResNets and VGG16. We make few observations: 1) low-rank mod-

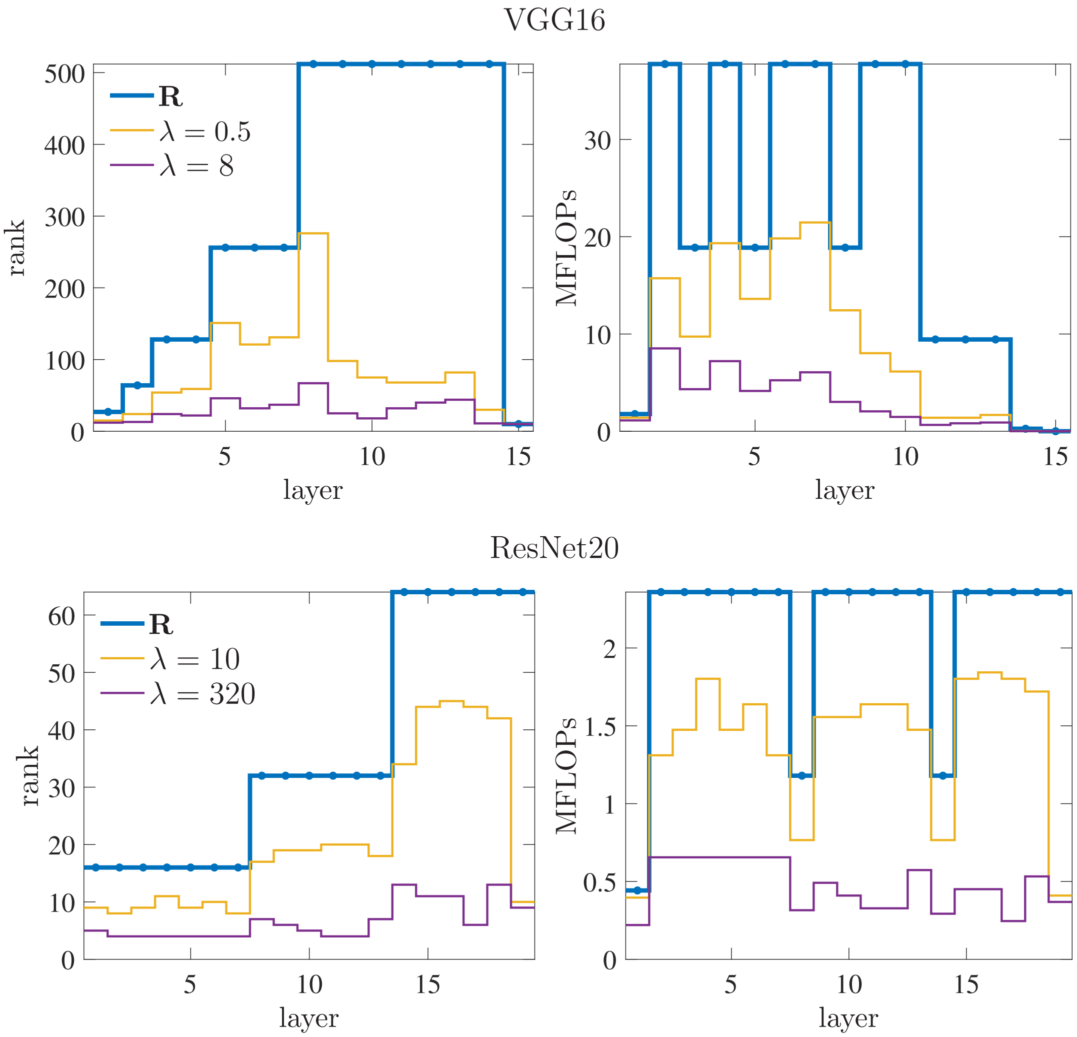


Figure 4.4: Some of final selected architectures in terms of rank and FLOPs of a layer for VGG16 and ResNet20 using our method. For λ values, the multiplicative factor of $\times 10^{-4}$ is omitted.

els obtained via our algorithm are *comparable and often considerably better* than other low-rank compression and structured pruning results 2) it is often beneficial in terms of error-FLOPs tradeoff to train a larger model and then compress it, for example, one of the low-rank VGG16-s with 107 MFLOPs achieve 6.11% error, which is comparable to the test error of the reference ResNet110 (6.02%) but with much fewer FLOPs (252 vs 107 MFLOPs).

As some portions of Figure 4.2 are a bit cluttered, we show separately the error-FLOPs tradeoffs for ResNet20, 32 and NiN in Fig 4.3. The rightmost plot in

Figure 4.2 compares our results to a joint weights-and-rank learning method of Li and Shi [80] called COBLA. Our method significantly outperforms COBLA with a higher margin in the high compression regime.

One question we need to ask is how selected ranks change over λ values? Or can we infer these ranks beforehand, fix them by decomposition and train with SGD, which will eliminate the need in joint weights-ranks learning altogether? We show selected ranks obtained by our method for each layer of the ResNet20 and VGG16, and corresponding FLOPs in Fig 4.4. We see that the selected ranks are not uniform at all, and some layers, e.g., layer 5 and 9 for VGG16, have much higher ranks comparing to others. Most importantly, their relative proportion does not stay the same for different λ values. Take a look at layers 10 and 11 of VGG16: for the value of $\lambda = 0.5$ the selected rank of layer 10 is greater of 11th, but for the higher value of $\lambda = 0.8$ the relation is reversed. These relations can not be captured by simple heuristics, and need to be inferred via joint optimization.

4.5 ImageNet experiments

We train the batch normalized version of the AlexNet network [75] having 62M parameters and 1140 MFLOPs on the ImageNet ILSVRC2012 dataset [108] using the augmentation procedure of the original paper. This network has a slightly larger FLOPs count (1170M vs 727M) when compared to Caffe-AlexNet due to not having group convolution. Our reference model achieves top-1 validation error of 40.43% and top-5 validation error of 17.55%. We compress the reference network using our rank-selection algorithm for both schemes 1 and 2 of low-rank decompositions using various λ-s.

We report our experimental results in Table 4.1. Although both decomposition schemes 1 and 2 allow us to achieve a significant FLOPs reduction while maintaining test errors close to the reference model — with scheme-1 we can get a model with 257 MFLOPs and top-1 validation error 40.81%, and with scheme 2 we can get a model with 166 MFLOPs with 40.46% top-1 error — we observe that scheme 2 is better suited for reducing the FLOPs count on AlexNet. For instance, 321

	$\lambda \times 10^{-4}$	# params.	MFLOPs	top-1 error, %	top-5 error, %
	R	62.3M	1139	40.43	17.55
scheme 1	0.05	43.0M	436	39.27	17.16
	0.15	17.6M	257	40.81	18.17
	0.17	15.8M	248	41.11	18.36
	0.20	13.8M	231	41.56	18.72
scheme 2	0.05	42.7M	321	**39.15**	**16.99**
	0.10	25.1M	226	49.60	17.40
	0.15	17.4M	185	39.93	17.47
	0.20	13.6M	166	40.46	17.71
	0.25	11.4M	**151**	41.03	18.23

Table 4.1: Our algorithm on AlexNet using low-rank parametrization schemes 1 and 2 (for several λs). We report: number of parameters and MFLOPs, and top-1/top-5 errors on the validation set (%).

MFLOPs version of scheme-2 AlexNet achieves a top-1 test error of 39.15% which is better than 436 MFLOPs version of scheme-1 AlexNet's test error of 39.27%.

In Table 4.2 we compare our low-rank AlexNet results to existing decomposition and structured pruning methods in the literature. Since most results in the literature use Caffe-AlexNet (which has smaller FLOPs count), we report FLOPs reduction ratios wrt to Caffe-AlexNet. Our compressed networks achieve *considerably better* speed-up ratios (ρ_{FLOPs}) and accuracies in comparison to other low-rank and filter pruning methods. Our smallest scheme-1 network has fewer FLOPs and better error than scheme-1 decomposed AlexNet of Wen et al. [120]. Our smallest scheme-2 network achieves 4.79× FLOPs reduction wrt Caffe-AlexNet while having the same accuracy, which outperforms similar scheme-2 methods of Kim et al. [69], Tai et al. [113], and structured pruning methods of Ding et al. [28], Li et al. [82], Yu et al. [133].

	MFLOPs	top-1 error, %	top-5 error, %	ρ_{FLOPs}
Caffe-AlexNet [67]	727	42.70	**19.80**	1.00
Kim et al. [70], Tucker	272	n/a	21.67	2.66
Tai et al. [113], scheme 2	185	n/a	20.34	3.90
Wen et al. [120], scheme 1	269	n/a	20.14	2.69
Kim et al. [69], scheme 2	272	43.40	20.10	2.66
Yu et al. [133], filter pruning	232	44.13	n/a	3.12
Li et al. [82], filter pruning	334	43.17	n/a	2.16
Ding et al. [28], filter pruning	492	43.83	20.47	1.47
ours, scheme 1, $\lambda = 0.20$	231	41.56	18.72	3.13
ours, scheme 2, $\lambda = 0.20$	**151**	**41.03**	18.23	**4.78**

Table 4.2: AlexNet compression with our algorithm vs published work using low-rank methods and structured pruning. We report top-1/top-5 validation error (%) and MFLOPs number and FLOPs reduciton ratio wrt Caffe-AlexNet.

Chapter 5

Device-targeted low-rank compression

For many tasks involving real-time audio/video processing and enhancement (e.g., speech to text, photo relighting) a too high inference time is unacceptable and may lead to the loss of customer base and profits. Different compression schemes have been proposed to address the inference time speed-up; however, most of the works handle it indirectly through a proxy optimization target: the total number of floating-point operations (FLOPs). While a smaller FLOPs count is indicative of a faster inference time, there is no one-to-one correspondence between smaller FLOPs and faster runtime. For example, on our testbed, the 727 MFLOPs version of the AlexNet (trained on ImageNet) runs a single image inference in 328 ms. In comparison, the CIFAR10 version of VGG16 has 314 MFLOPs, which is 2.32× fewer than AlexNet; yet, it runs 6.07× faster (54 ms) illustrating that on-device runtime does not only depend on the total FLOPs. Indeed, the inference runtime is the function of the neural network's overall structure and the hardware characteristics (e.g., frequency of CPU/GPU, size of the cache, memory speed), and it cannot be extrapolated from a single FLOPs-count number.

We consider the problem of inference-targeted compression of a neural network for a given device and adopt the low-rank compression as our method of choice. While such a scheme has a history of usage for network compression problems to reduce FLOPs and size of the networks, we show that it can be effectively used to directly target the on-device inference time of compressed models due to the following. First, as we discuss in section 5.1, the low-rank scheme gives rise to a simple yet accurate device-runtime model that can be used to a precise estimation of the actual inference time of the compressed model. Second, the computational reductions of low-rank compression are realizable without specific hardware support (unlike, for instance, elementwise pruning [38]): if the layer with weights $\mathbf{W}$ is compressed with r-rank matrix $\mathbf{U}\mathbf{V}^T$, then forward pass of $\mathbf{W}\mathbf{x}$ through that layer can be implemented as a forward pass through a sequence of layers with weights $\mathbf{V}^T$ and $\mathbf{U}$.

The problem we are solving is challenging: we need to find the best configuration of ranks (one rank per layer, integer values) and corresponding low-rank weights (floating-point values) so that network has the fastest on-device inference time while maintaining its original task performance. Assuming we have K layers with M possible ranks per layer, the problem involves selection over the set of M^K distinct rank configurations. However, as we show in section 5.2, a suitable formulation of this problem using the proposed device-runtime model admits an efficient algorithm involving alternation of simple steps: a step over weights of the neural networks (solved by stochastic gradient descent, SGD) and a step over the rank configurations (solved by enumeration). In section 5.4 we experimentally validate our approach's effectiveness by compressing the AlexNet and VGG16 to have fast inference time on the ARM Cortex-A57 CPU of the NVIDIA's Jetson Nano embedded computing platform.

Related work Several works use the resulting number of FLOPs as an optimization criterion when optimizing over the ranks [53, 55, 56, 80]. However, we are not aware of any methods that directly optimize the on-device inference speed.

5.1 Device runtime model

Assume we are given a neural network with K layers and the weights $\mathbf{W} = \{\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K\}$ where $\mathbf{W}_k$ is a weight matrix (or tensor) of the kth layer. The weights $\mathbf{W}$ implicitly define a computational graph for an inference pass through

CPU	Quad-core ARM Cortex-A57, 1.4 GHz
GPU	128 CUDA cores at 0.9 GHz
RAM	4 GB 64-bit LPDDR4, 1.6 GHz
OS	Ubuntu 18.04.5 LTS
Kernel	GNU/Linux 4.9.140-tegra
Storage	128 GB microSDXC memory card
Software	PyTorch v1.6.0, ONNXRuntime v1.4.0

Table 5.1: Specifications of NVIDIA's Jetson Nano Developer kit used as our target testbed. While it has a built-in GPU, we used the CPU inference time (parallelized on two threads) as our compression goal.

the network. When we execute this graph on the given hardware, we can measure the inference time. Throughout this chapter, we define the inference time as the total time required to complete a forward pass of a single image through the computational graph, and call it $\mathcal{R}(\mathbf{W})$.

In our model, we assume that the total inference time $\mathcal{R}(\mathbf{W})$ is the sum of the inference times through each of the K layers since layers have to be processed sequentially:

$$\mathcal{R}(\mathbf{W}) = \mathcal{R}_1(\mathbf{W}_1) + \mathcal{R}_2(\mathbf{W}_2) + \cdots + \mathcal{R}_K(\mathbf{W}_K). \tag{5.1}$$

Here, each of the $\mathcal{R}_k(\mathbf{W}_k)$ measures the total inference time through a layer k: this involves the time to load the weights and the inputs, actual computation time, and time to unload the output. In reality, the right hand side of eq. (5.1) is an upper bound to the total runtime $\mathcal{R}(\mathbf{W})$: when the computational graph is executed optimally, some weights and inputs can be prefetched and layer-to-layer computations can be pipelined, thus, finishing earlier than the sum of separate inferences through each layer.

When we compress the network using the low-rank decomposition, the kth layer is compressed by an r_k-rank matrix, and the forward pass through the layer can be implemented as a sequence of fully-connected or convolutional layers (sec. 2.1.4). Since the computational graph is defined by the shape of the weight matrices, and not by the weight values, we conclude that the inference time through a layer k is

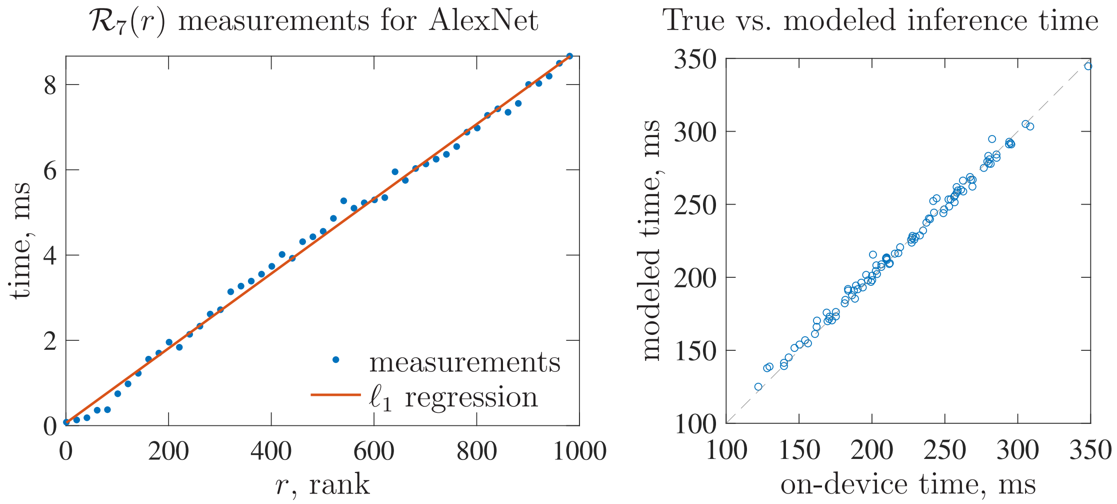


Figure 5.1: *Left:* Measurements and regression fit to model the inference time as a function of rank for the 7th layer of AlexNet. *Right:* Plot of the actual, on device inference time for 100 randomly sampled low-rank configurations of AlexNet vs. the predictions of our model $\mathcal{R}(\mathbf{r})$. On these samples, the mean average error was 3.03 ms.

a function of the rank, and our model simplifies as:

$$\mathcal{R}(\mathbf{W}) = \mathcal{R}(\mathbf{r}) = \mathcal{R}_1(r_1) + \mathcal{R}_2(r_2) + \cdots + \mathcal{R}_K(r_K). \tag{5.2}$$

We make several observations. First, due to a small number of possible ranks per layer, we can directly measure the value of $\mathcal{R}_k(r)$ on the device. Essentially, $\mathcal{R}_k(r)$ is a lookup table with a single measurement for each r. Second, the proposed model is computationally efficient and avoids a combinatorial number of measurements. Assuming there are M possible ranks per layer ($r_k = 1, \ldots, M$), rather than making M^K measurements for all possible rank configuration we only need MK on-device measurement in total.

Even though we need to consider M ranks per layer, collecting the inference times might be time consuming and impractical. Particularly, the measurements need to be repeated many times to reduce the noise, however, too many measurements at a time might induce the thermal throttling[1] of the target device which adds inconsistencies to the model, and measurements need to be taken at intermittent intervals.

[1] `https://en.wikipedia.org/wiki/CPU_throttling`

Due to the aforementioned considerations, in the actual implementation of the proposed model we collected the low-rank inference measurements at certain rank intervals and then fit a regression curve. To make the measurements, we use highly-optimized implementation of the forward pass through the ONNX runtime. When we used the CPU of Jetson Nano Developer board as our target device (see Table 5.1), we noticed that measurements within each layer follow a line trend except for some outliers (which are presumably caused by noise). Therefore, we computed an ℓ_1-fit and used the fitted lines as our $\mathcal{R}_k$ functions (see left of Figure 5.1). In our experiments, we found that ℓ_1-fitted regression can model rank-dependent device runtime pretty accurately across all layers.

How good is our model? To answer this question, we sampled random rank configurations for the AlexNet architecture by choosing each layer's rank uniformly (out of possible ones) and measured the true inference speeds of the sampled architectures. On the right of Figure 5.1 we compare true inference times to the modeled inference times. As we can see, the difference between our model and the true inference time is minuscule: the average difference on the sampled low-rank architectures was 3.03 milliseconds.

5.2 Problem formulation and optimization

Having developed the device runtime model $\mathcal{R}(\mathbf{r})$ for a given K-layer network with weights $\mathbf{W} = \{\mathbf{W}_1, \dots, \mathbf{W}_K\}$, now we give a low-rank compression formulation that targets the inference time on the given device. We denote the network's task loss (e.g., cross-entropy) as L and define the following optimization problem of

$$\min_{\mathbf{W},\mathbf{r}} \quad L(\mathbf{W}) + \lambda \mathcal{R}(\mathbf{r}) \quad \text{s.t.} \quad \operatorname{rank}(\mathbf{W}_k) = r_k,\ k = 1, \dots, K, \tag{5.3}$$

where the term $\lambda \mathcal{R}(\mathbf{r})$ with user-chosen $\lambda > 0$ controls the amount of desired reduction of the inference time.

The problem given by eq. (5.3) is a mixed-integer optimization involving the floating-point weights of the neural network and the integer rank values. Typically, even the neural network part on its own (without the rank constraints) requires many iterations over the training dataset to be properly optimized (with SGD), and combination with rank constraints makes it truly challenging. Fortunately, this formulation falls into the category of *model compression as constrained optimization* problems [15] and admits an efficient solution based on the *Learning-Compression* (LC) algorithm.

To derive the LC algorithm corresponding to our formulation, let us equivalently rewrite (as in chapter 4) the constraints by introducing the auxiliary variables $\boldsymbol{\Theta}_k$ for each $k = 1, \dots, K$ as

$$\operatorname{rank}(\mathbf{W}_k) = r_k \iff \mathbf{W}_k = \boldsymbol{\Theta}_k,\ \operatorname{rank}(\boldsymbol{\Theta}_k) = r_k,$$

and then apply penalty method [101, ch.17] to the matrix terms (i.e., $\mathbf{W}_k = \boldsymbol{\Theta}_k$) while driving $\mu \to \infty$ (norms are Frobenius):

$$\begin{aligned} \min_{\mathbf{W},\boldsymbol{\Theta},\mathbf{r}} \quad & L(\mathbf{W}) + \frac{\mu}{2}\sum_{k=1}^{K} \|\mathbf{W}_k - \boldsymbol{\Theta}_k\|^2 + \lambda\mathcal{R}(\mathbf{r}) \\ \text{s.t.} \quad & \operatorname{rank}(\boldsymbol{\Theta}_k) = r_k,\ k = 1, \dots, K. \end{aligned} \tag{5.4}$$

We use the quadratic penalty to simplify the derivations; however, in practice, we use the augmented Lagrangian method, which has an additional step over the vector of Lagrange multipliers. If we apply alternating optimization over variables $\mathbf{W}$ and $\{\boldsymbol{\Theta}, \mathbf{r}\}$ we obtain the substeps that can be efficiently handled:

- Learning (L) step. The step over $\mathbf{W}$ has the form of:

  $$\min_{\mathbf{W}} L(\mathbf{W}) + \frac{\mu}{2}\sum_{k=1}^{K} \|\mathbf{W}_k - \boldsymbol{\Theta}_k\|^2.$$

- Compression (C) step: The step over $\boldsymbol{\Theta}$ and $\mathbf{r}$ separates into K smaller substeps due to the form of $\mathcal{R}$ (eq. 5.2):

  $$\min_{\boldsymbol{\Theta}_k, r_k} \quad \frac{\mu}{2}\|\mathbf{W}_k - \boldsymbol{\Theta}_k\|^2 + \lambda\mathcal{R}_k(r_k) \quad \text{s.t.} \quad \operatorname{rank}(\boldsymbol{\Theta}_k) = r_k.$$

```
input $K$-layer neural net with weights $\mathbf{W} = \{\mathbf{W}_1, \dots, \mathbf{W}_K\}$,
      hyperparameter $\lambda$, device runtime model $\mathcal{R}$.
$\mathbf{W} = (\mathbf{W}_1, \dots, \mathbf{W}_K) \leftarrow \arg\min_{\mathbf{W}} L(\mathbf{W})$          reference net
$\mathbf{r} = (r_1, \dots, r_K) \leftarrow \mathbf{0}$          ranks
$\boldsymbol{\Theta} = (\boldsymbol{\Theta}_1, \dots, \boldsymbol{\Theta}_K) \leftarrow \mathbf{0}$          auxillary variables
for $\mu = \mu_1 < \mu_2 < \dots < \mu_T$
  $\mathbf{W} \leftarrow \arg\min_{\mathbf{W}} L(\mathbf{W}) + \frac{\mu}{2} \sum_{k=1}^{K} \|\mathbf{W}_k - \boldsymbol{\Theta}_k\|^2$          L step
  for $k = 1, \dots, K$          C step
    $\boldsymbol{\Theta}_k, r_k \leftarrow \arg\min_{\boldsymbol{\Theta}_k, r_k} \frac{\mu}{2} \|\boldsymbol{\Theta}_k - \mathbf{W}_k\|^2 + \lambda \mathcal{R}_k(r_k)$
  if $\|\mathbf{W} - \boldsymbol{\Theta}\|$ is small enough then exit the loop
return $\mathbf{W}, \boldsymbol{\Theta}, \mathbf{r}$
```

Figure 5.2: LC algorithm to jointly learn weights and ranks when applying the low-rank compression to target on-device inference speed.

5.3 Solutions of L and C steps

The L-step problem is a standard neural network training (learning) with added ℓ_2 regularization. We solve it using SGD. The C-step problem can be interpreted as finding best low-rank approximation (compression) to the matrix $\mathbf{W}_k$ in the presence of a cost function over the ranks. The solution of this problem was given in section 4.2 and requires computing a singular value decomposition of $\mathbf{W}_k$ followed by enumeration.

Overall, the LC algorithm alternates between L and C steps while driving $\mu \to \infty$. The L step finds (locally) optimal weights $\mathbf{W}$ that are close to the current selection of the low-rank matrices ($\boldsymbol{\Theta}$) with the rank configuration $\mathbf{r}$. The C step finds the best configuration of the ranks and the optimal numeric values of the low-rank matrices that approximate the current weights $\mathbf{W}$. Once μ is sufficiently large, neural network weights $\mathbf{W}$ and its compressed form $\boldsymbol{\Theta}$ will reach equality by satisfying $\mathbf{W}_k = \boldsymbol{\Theta}_k$.

	Model	MFLOPs	Inference time	top-1 err	top-5 err
	reference (**R**)	1140	378.5 ms	40.43%	17.55%
	Caffe-AlexNet [67, 75]	727	328.7 ms	42.90%	19.80%
ours	$\lambda = 5.0 \times 10^{-3}$	421	104.1 ms	**38.88%**	**16.83%**
	$\lambda = 1.0 \times 10^{-2}$	290	69.2 ms	39.12%	17.03%
	$\lambda = 2.0 \times 10^{-2}$	186	**42.0 ms**	40.34%	17.64%
	low-rank AlexNet (ch. 4)	227	83.6 ms	39.61%	17.40%
	low-rank AlexNet (ch. 4)	**166**	50.2 ms	40.46%	17.71%
	ENC-AlexNet [69]	272	93.3 ms	43.40%	19.93%
	SqueezeNet 1.1 [50]	352	63.8 ms	42.90%	19.70%

Table 5.2: Details of selected low-rank AlexNets obtained with our algorithm, and comparison to some of the available low-rank AlexNets in the literature. We additionally include a comparison to the SqueezeNet [50] that has similar accuracy to the AlexNet but was manually designed to be small and fast. All reported runtime measurements are performed on our testbed: CPU of Jetson Nano.

5.4 Experiments

We demonstrate the effectiveness of our approach by compressing batch normalized versions of AlexNet (trained on ImageNet) and VGG16 (trained on CIFAR10) networks. We initialize the algorithm from the reasonably well-trained reference models. Our reference AlexNet has 62.3M parameters, 1140 MFLOPs, and the top-1/top-5 validation error of 40.43%/17.55%. We did not use group convolutions in our reference version of AlexNet, therefore it has a slightly larger FLOPs count of 1140 MFLOPs, whereas standard (Caffe-version) has 727 MFLOPs [67, 75]. The reference CIFAR10 VGG16 model has 15.3M parameters, 313.73 MFLOPs, and a test error of 6.46%.

As our target device we use the ARM Cortex-A57 CPU of the NVIDIA's Jetson Nano; full specifications are available in Table 5.1. Single image inference times on this CPU (using two threads) are 378.45 ms for AlexNet and 53.99 ms for VGG16. For each network we build the runtime model as specified in section 5.1. The weights of the convolutional layers are reshaped using the scheme 2 (section 2.1.4).

We run our LC algorithm for T steps with an exponential schedule on μ with $\mu_t = a \times b^t$ at the tth step: for AlexNet we set $T = 30, a = 10^{-4}$ and $b = 1.2$; for VGG16: $T = 60, a = 10^{-5}$ and $b = 1.2$. Each L step was trained with stochastic gradient descent using the following settings: for AlexNet we used the learning rate of 0.001 (decayed by 0.9 after each epoch) with the momentum of 0.9 on minibatches of 256 images; for VGG16 we used the learning rate of 7×10^{-4} (decayed by 0.99 after each epoch) with the momentum of 0.9 on minibatches of size 128 images. Each L step used a predetermined number of epochs (i.e., full passes over the dataset): 5 epochs for AlexNet and 20 epochs for VGG16. Once the algorithm finished, we finetuned the decomposed weights for a small number of epochs (AlexNet: 30 epochs, VGG16: 100 epochs). *Overall, the entire compression pipelines requires not more than* 2.5× *the time required to train the reference networks in the first place.*

To explore the error-compression tradeoff, we run our compression with various values of λ. We report our results in Figure 5.3 as inference time vs. validation error over the range of the obtained networks. To put our result in perspective, we additionally plot the results of FLOPs guided low-rank compression of [53].

For both networks, we achieve lower test error for the same inference speed when compared to the results of FLOPs guided low-rank compression. Notably, with $\lambda = 1 \times 10^{-2}$ we obtain a low-rank AlexNet model that has the validation error of 39.12% and requires only 69.2 ms to complete its inference pass on our target device. This results in a speed-up of 5.47× wrt reference model and 4.74× wrt Caffe-AlexNet while having 1.5% improvement in the test error wrt reference. The FLOPs count of this particular network is not that small: it requires 290 MFLOPs of compute; and compressed AlexNets with fewer FLOPs are available in the literature (see Table 5.2). However, the architecture of our compressed network was directly optimized to run as fast as possible on the target device, therefore, even with 290 MFLOPs it runs faster than the 227 MFLOPs low-rank AlexNet of [53] and the 272 MFLOPs low-rank AlexNet of [69], while additionally having a better accuracy.

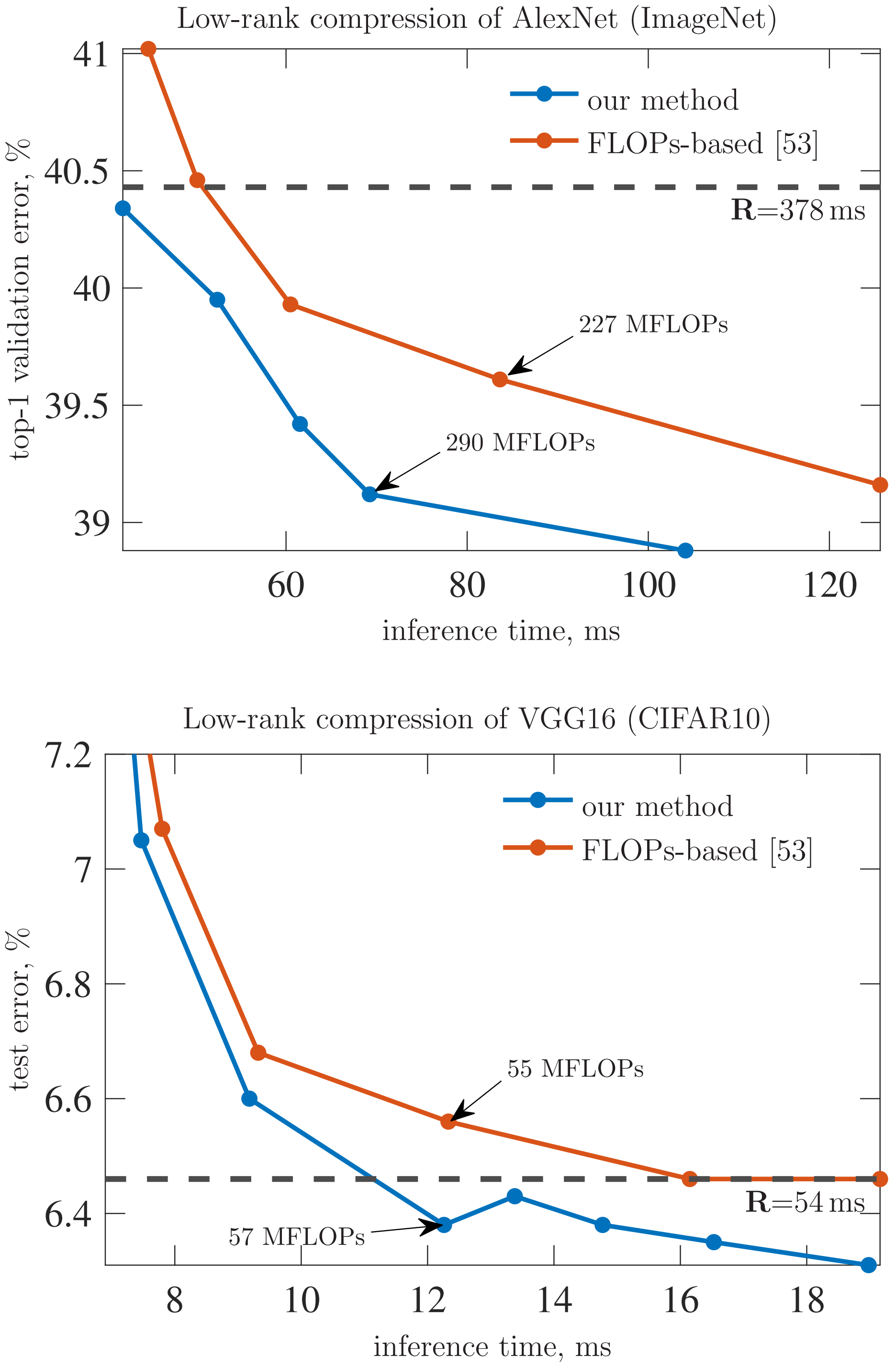


Figure 5.3: Inference speed vs. error plot for our (blue) compressed AlexNet (top) and VGG16 models (bottom); for both networks, we additionally compare to the FLOPs based low-rank compression of [53] (given with red). The test errors and inference times of the reference models are indicated by horizontal dashed line labeled as **R**.

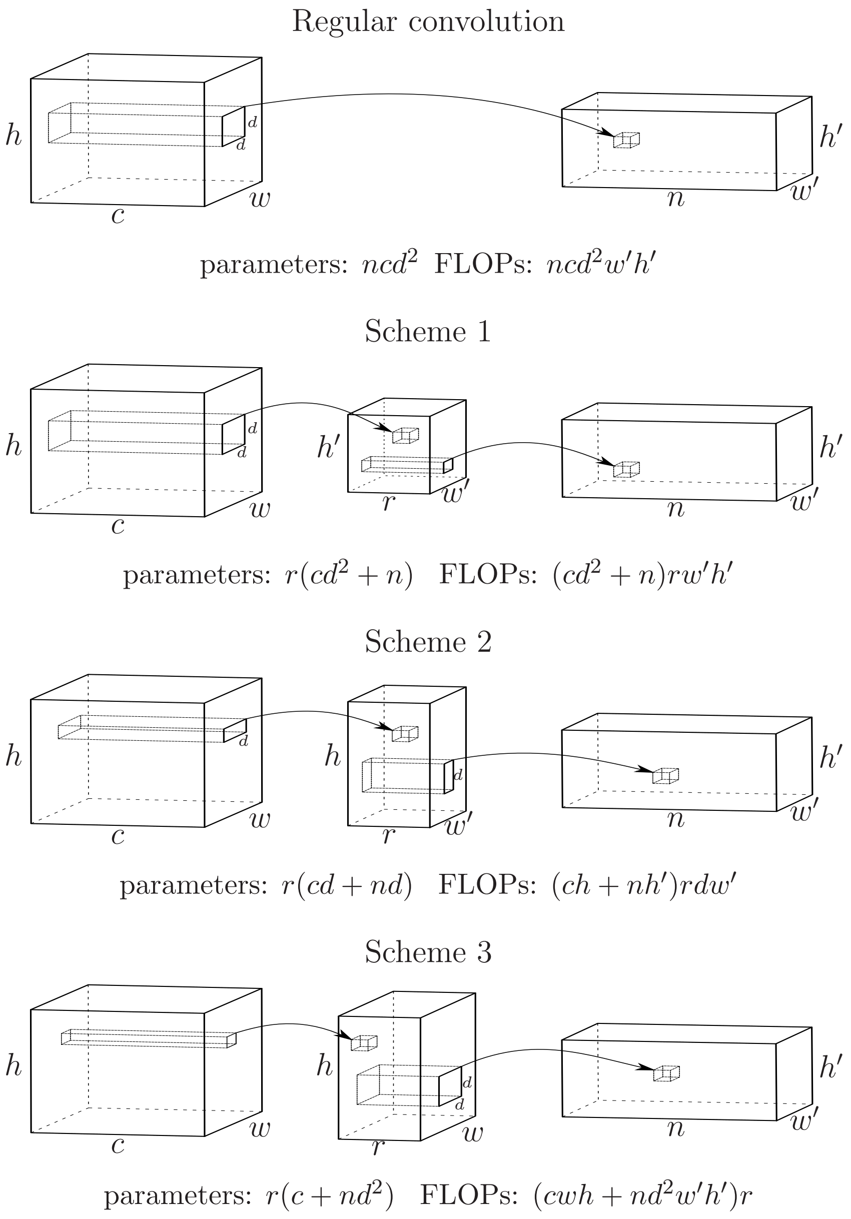
We see a similar pattern for VGG16 results. For instance, with $\lambda = 1.4 \times 10^{-2}$ our algorithm achieves a network that requires only 12.26 ms of CPU time (4.40× faster) while having a test error of 6.38%. This network has a total of 57.3 MFLOPs, yet, it runs faster than 55.3 MFLOPs low-rank VGG16 of [53]: 12.26 ms vs. 12.33 ms.

Chapter 6

Low-rank compression of convolutional layers: decomposition schemes

When applying the low-rank methods, we decompose the weight matrix as a product $\mathbf{U}\mathbf{V}^T$ of lower rank matrices. For fully connected layers, where weights are naturally in a matrix form, this parametrization is straightforward to apply. However, the weights of the convolutional layers come as tensors; therefore, to apply a low-rank, we should first reshape its weights into a matrix. Formally, a *matrix reshape*is a reordering of the items in a tensor $\mathcal{A}$ into a matrix $\mathbf{A}$ so that matrix $\mathbf{A}$ contains the same set of items as $\mathcal{A}$. There are many possible matrix reshapes of a tensor, and each reshape gives a rise to a different *low-rank decomposition scheme*. A few of the decomposition schemes can be implemented as a sequence of convolutional layers, allowing to harness the compressive and speeding-up properties of low rank; for a detailed overview of these schemes we refer to section 2.1.4.

In previous low-rank compression works, the decomposition scheme (e.g., scheme 1) was fixed and applied throughout the network. This is suboptimal in practice, as each scheme has its own advantages and should be selected accordingly, per layer. To address this, we want to select the best decomposition scheme for every layer of a given neural network. One simple but not an efficient solution to this



Regular convolution

parameters: ncd^2 FLOPs: $ncd^2w'h'$

Scheme 1

parameters: $r(cd^2+n)$ FLOPs: $(cd^2+n)rw'h'$

Scheme 2

parameters: $r(cd+nd)$ FLOPs: $(ch+nh')rdw'$

Scheme 3

parameters: $r(c+nd^2)$ FLOPs: $(cwh+nd^2w'h')r$

Figure 6.1: Illustration of a regular convolution operation (top left) and its rank-r decompositions according to schemes 1, 2, and 3. The input to a layer has the shape of $c \times w \times h$ and is depicted as a cube with the appropriately marked dimensions. When the input is convolved with n filters of dimension $c \times d \times d$ (filters are not shown) it generates an output tensor of shape $n \times w' \times h'$. Each arrow represents a convolution of a portion of the input with a single filter, and points to the result of this convolution, a cube of size $1 \times 1 \times 1$. Low-rank decomposition schemes replace the convolutional layer with a sequence of two convolutions.

selection problem is to try all possible combinations of decompositions using an off-the-shelf low-rank compression algorithm. For a K layer neural network with M different decomposition schemes to try, the total number of combinations is M^K: this number of trials is unmanageable with the average depth of modern neural networks having dozens of layers (e.g., ResNet-152 has $K = 152$ layers).

The problem exacerbates when we include the rank selection problem: clearly, the performance of the compressed network is a function of the rank as well as the decomposition scheme. How can we select the ranks and the decompositions schemes for every layer of the neural network to fit into our constraints yet avoiding the associated combinatorial explosion? We approach this problem by formulating a model selection problem that captures both rank and shape selection as part of the objective. We then show that our formulation is amenable to the alternating optimization and give an efficient algorithm to learn ranks, shapes, and weights of the neural network.

6.1 Problem formulation

Assume we are given a K-layer neural network trained to minimize a task loss L (e.g., cross-entropy) over its weights $\mathbf{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_K\}$ where the $\mathcal{W}_k$ is the weight tensor of the layer k. Let as denote the matrix reshape of the tensor $\mathcal{W}_k$ that induces the low-rank decomposition scheme s as $\mathcal{R}(\mathcal{W}_k, s)$, and the actual reshaped matrix as $\mathbf{\Theta}_k = \mathcal{R}(\mathcal{W}_k, s)$. We want to select the best scheme and rank for each layer to optimize the tradeoff between the model loss and a compression cost $\mathcal{C}(\mathbf{\Theta}, \mathbf{r})$. To achieve this goal, we impose the low-rank structure on the reshaped weights of each layer via explicit rank constraints on the corresponding $\mathbf{\Theta}_k$-terms, and form the following model selection problem over the ranks $\mathbf{r} = \{r_1, \ldots, r_K\}$, decomposition schemes $\mathbf{s} = \{s_1, \ldots, s_K\}$, and weights $\mathbf{W}$:

$$\begin{aligned} \min_{\mathbf{W}, \mathbf{\Theta}, \mathbf{r}, \mathbf{s}} \quad & L(\mathbf{W}) + \lambda\, \mathcal{C}(\mathbf{\Theta}, \mathbf{r}) \\ \text{s.t.} \quad & \mathbf{\Theta}_k = \mathcal{R}(\mathcal{W}_k, s_k), \\ & \operatorname{rank}(\mathbf{\Theta}_k) = r_k, \quad \forall k = 1, \ldots, K \end{aligned} \tag{6.1}$$

We control the amount of the compression (and subsequent tradeoff) via a parameter $\lambda > 0$; and the compression cost function $\mathcal{C}(\boldsymbol{\Theta}, \mathbf{r})$ will encourage having smaller models. It is up to the user to determine the optimal operating point wrt λ: usually multiple values are considered to select among a family of compressed models (see Figure 6.3). We define the $\mathcal{C}(\boldsymbol{\Theta}, \mathbf{r})$ to be layerwise separable function:

$$\mathcal{C}(\boldsymbol{\Theta}, \mathbf{r}) = \mathcal{C}(\boldsymbol{\Theta}_1, r_1) + \cdots + \mathcal{C}(\boldsymbol{\Theta}_K, r_K). \tag{6.2}$$

Such a cost function can handle multiple targets of interest:

- It can target the storage and FLOPs of the compressed model, as both of these are the functions of the rank and can be written as $C(\boldsymbol{\Theta}_k, r_k) = \alpha \times r_k$ for some constant α (see chapters 4–5).
- It can target the nuclear norm [31] of weight matrices instead: $\mathcal{C}(\boldsymbol{\Theta}_k, r_k) = \|\boldsymbol{\Theta}_k\|_*$. Such penalty has been well studied in the compressed sensing field and known to have low-rank inducing properties.

6.2 Optimization algorithm

The problem (6.1) is discrete over the ranks, schemes, and reshapes, but continuous over the weights, which makes it a challenging optimization problem. Fortunately, the formulation of (6.1) is in *learning-compression* form [15] which admits alternating optimization solution [15, 16, 17, 53, 54, 55]. To obtain the algorithm, we equivalently reformulate the problem (6.1) using the quadratic penalty [101]. (Here we use quadratic penalty for brevity of presentation. In practice we use augmented Lagrangian version which has an additional step over the Lagrange multipliers.) We apply the penalties only to the reshaping constraints of $\boldsymbol{\Theta}_k = \mathcal{R}(\mathcal{W}_k, s_k)$ and optimize the following while driving $\mu \to \infty$:

$$\begin{aligned} \min_{\mathbf{W}, \boldsymbol{\Theta}, \mathbf{r}, \mathbf{s}} \quad & L(\mathbf{W}) + \lambda \mathcal{C}(\boldsymbol{\Theta}, \mathbf{r}) + \frac{\mu}{2} \sum_{k=1}^{K} \|\boldsymbol{\Theta}_k - \mathcal{R}(\mathcal{W}_k, s_k)\|_F^2 \\ \text{s.t.} \quad & \operatorname{rank}(\boldsymbol{\Theta}_k) = r_k, \quad \forall k = 1, \ldots, K. \end{aligned} \tag{6.3}$$

The reformulation (6.3) allows us to efficiently optimize the problem by alternating over $\mathbf{W}$ and $\{\boldsymbol{\Theta}, \mathbf{r}, \mathbf{s}\}$. This results into learning (L) and compression (C) steps:

- **L step**: $\min_{\mathbf{W}} L(\mathbf{W}) + \frac{\mu}{2}\sum_{k=1}^{K} \|\mathbf{\Theta}_k - \mathcal{R}(\mathcal{W}_k, s_k)\|_F^2$
  The step over $\mathbf{W}$ is fully differentiable, and has a simple ℓ_2-regularized form of the neural network training. We will use SGD to solve this step.

- **C step**: $\min_{\mathbf{\Theta},\mathbf{r},\mathbf{s}} \lambda\,\mathcal{C}(\mathbf{\Theta}, \mathbf{r}) + \frac{\mu}{2}\sum_{k=1}^{K} \|\mathbf{\Theta}_k - \mathcal{R}(\mathcal{W}_k, s_k)\|_F^2$
  The step over $\mathbf{\Theta}, \mathbf{r}$ and $\mathbf{s}$ is still a mixed-integer optimization problem, however, it admits an efficient solution depending on the form of compression cost $\mathcal{C}$.

The alternation of L and C steps guarantee a monotonic decrease of the objective function. More importantly, it confines the combinatorial search over the ranks and decomposition schemes to a subproblem that does not involve the network loss (which typically requires iteration over a large dataset).

Solution of the C step The layerwise separable cost function (6.2) splits the C-step problem into subproblems over each layer k:

$$\begin{aligned} \min_{\mathbf{\Theta}_k, r_k, s_k} \quad & \lambda\,\mathcal{C}(\mathbf{\Theta}_k, r_k) + \frac{\mu}{2}\|\mathbf{\Theta}_k - \mathcal{R}(\mathcal{W}_k, s_k)\|_F^2 \\ \text{s.t.} \quad & \operatorname{rank}(\mathbf{\Theta}_k) = r_k. \end{aligned} \tag{6.4}$$

For a fixed decomposition scheme s_k the solution of this optimization problem over $\mathbf{\Theta}_k, r_k$ is known in closed form for multiple costs $\mathcal{C}$. For the storage and FLOPs costs, the solution involves SVD and enumeration over the ranks as was shown in section 4.2. For the nuclear-norm cost, the solution involves singular value shrinkage [12]. Therefore, to find the solution of (6.4) we iterate over possible schemes, and select the triplet $(\mathbf{\Theta}_k, r_k, s_k)$ attaining the minimum loss of eq. (6.4). See Figure 6.2 for the full pseudocode.

6.3 Experimental evaluation and discussion

We demonstrate the power of jointly training weights, ranks, and decompositions schemes by compressing various models on different datasets. We compress the Caffe version of LeNet5 on MNIST dataset, batch normalized VGG16

```
input $K$-layer neural net with weights $\mathbf{W} = \{\mathcal{W}_1, \dots, \mathcal{W}_K\}$,
      hyperparameter $\lambda$, cost function $\mathcal{C}$,
      set of reshaping schemes $\{\mathcal{S}_1, \dots, \mathcal{S}_m\}$
$\mathbf{W} = (\mathcal{W}_1, \dots, \mathcal{W}_K) \leftarrow \arg\min_{\mathbf{W}} L(\mathbf{W})$        reference net
$\mathbf{s} = (s_1, \dots, s_K) \leftarrow (\mathcal{S}_1, \dots, \mathcal{S}_1)$        decomposition schemes
$\mathbf{r} = (r_1, \dots, r_K) \leftarrow \mathbf{0}$        ranks
$\mathbf{\Theta} = (\mathbf{\Theta}_1, \dots, \mathbf{\Theta}_K) \leftarrow \mathbf{0}$        reshaped weights
for $\mu = \mu_1 < \mu_2 < \dots < \mu_T$
  $\mathbf{W} \leftarrow \arg\min_{\mathbf{W}} L(\mathbf{W}) + \frac{\mu}{2} \sum_{k=1}^{K} \|\mathbf{\Theta}_k - \mathcal{R}(\mathcal{W}_k, s_k)\|^2$        L step
  for $k = 1, \dots, K$        C step
    for $s'_k = \mathcal{S}_1, \dots, \mathcal{S}_m$
      $\mathbf{\Theta}'_k, r'_k \leftarrow \arg\min_{\mathbf{\Theta}_k, r_k} \lambda \mathcal{C}_k(r_k) + \frac{\mu}{2} \|\mathbf{\Theta}_k - \mathcal{R}(\mathcal{W}_k, s'_k)\|^2$
      if $(\mathbf{\Theta}'_k, r'_k, s'_k)$ has a lower C-step objective then
        $(\mathbf{\Theta}_k, r_k, s_k) \leftarrow (\mathbf{\Theta}'_k, r'_k, s'_k)$
  return $\mathbf{W}, \mathbf{\Theta}, \mathbf{r}$
```

Figure 6.2: Pseudocode of the LC algorithm to jointly learn weights, ranks, and low-rank decomposition schemes to compress a network

on CIFAR10, and AlexNet on ImageNet. While our algorithm can handle different compression costs $\mathcal{C}$, we run our experiments with $\mathcal{C}$ targeting the resulting MFLOPs reduction of the models. Our algorithm is initialized from reasonably well pre-trained reference models and run with different values of λ to explore the entire error-FLOPS tradeoff space. We allow the algorithm to select over schemes 1, 2, and 3 of sec 2.1.4. Overall, the total runtime of compression does not take more than 3× the time spend on training the reference model in the first place. We run L and C steps in a total of T times with the μ value of $\mu_{\text{init}} \times b^t$ at step t, and perform finetuning for 10–20 epochs afterwards. All L steps are optimized using SGD with a momentum of 0.9 and the initial learning rate is decayed by 0.99 after each epoch. The exact values are as follows:

λ	The selected scheme and rank over layers															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2.0×10^{-5}	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_3$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	31	11	9
	16	32	71	97	116	238	263	254	292	172	122	99	105			
7.5×10^{-5}	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	$\mathcal{S}_2$	23	10	9
	15	20	43	53	113	110	116	239	124	79	72	74	89			

Table 6.1: The final selected ranks and reshaping schemes (across layers) for some of the compressed VGG16 models on CIFAR10 using our algorithm. The VGG16 network has 16 layers: layers 1–13 are convolutional, for which we selected both schemes and ranks, and layers 14–16 are fully connected, for which we select only the ranks. We denote schemes as: $\mathcal{S}_1$ (scheme 1), $\mathcal{S}_2$ (scheme 2), $\mathcal{S}_3$ (scheme 3). Notice how selected scheme and ranks change when we use a higher value of λ, e.g., the selected scheme for layer 5 changed from $\mathcal{S}_3$ to $\mathcal{S}_2$ The reference VGG16 has the test error of 6.45% with 317 MFLOPs and 15.2M parameters, the low-rank VGG16 with $\lambda = 2 \times 10^{-5}$ (first row) has the test error of 5.90%, 156 MFLOPs, and 4.8M parameters, the low-rank VGG16 with $\lambda = 7.5 \times 10^{-5}$ (second row) has the test error of 5.97%, 78 MFLOPs, and 2.5M parameters.

- **LeNet5**: $T = 30$, $\mu_{\text{init}} = 0.001$, $b = 1.1$. Each L step runs for 30 epochs with a learning rate of 0.02.
- **VGG16**: $T = 60$, $\mu_{\text{init}} = 0.0002$, $b = 1.2$. Each L step runs for 15 epochs with a learning rate 0.0001.
- **AlexNet**: $T = 30$, $\mu_{\text{init}} = 0.001$ and $b = 1.1$. Each L step is run for 15 epochs with a learning rate 0.0005.

We plot our rank-and-scheme-optimized LeNet5 and VGG16 models on Figure 6.3. To give a perspective on whether the scheme selection improves the overall compression, we additionally run our algorithm with a fixed reshape (using schemes 1, 2, or 3) throughout the net, effectively disabling the scheme selection. As expected, low-rank networks trained with only a fixed scheme do not achieve competitive error-FLOPs tradeoff when compared to the scheme optimized counterparts. For instance, our scheme optimized low-rank LeNet5-s have no accuracy loss up to 0.6 MFLOPs, which corresponds to ×4.25 speed-up; and our com-

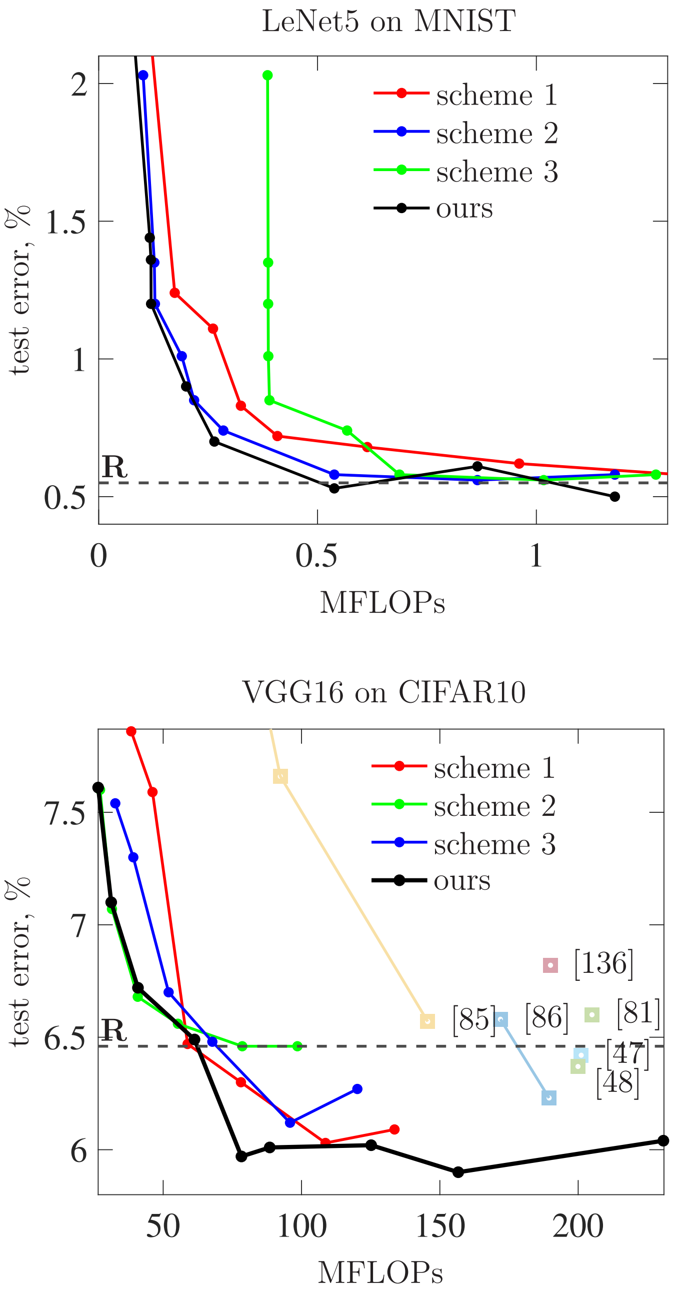


Figure 6.3: Compression of LeNet5 and VGG16 networks trained on MNIST and CIFAR10 datasets using automatic rank selection with fixed decomposition schemes 1, 2, 3 and comparison to our approach where schemes and ranks are learned jointly. For each scheme (and our method) we run multiple compression and generate a family of model which we plot as a curve. We additionally plot some of the available compression results in the literature using square markers. Horizontal dashed lines marked with **R** indicate the test-error of reference (uncompressed) networks.

	MFLOPs	top-1 error, %	top-5 error, %
Caffe-AlexNet	727	42.70	19.80
Tai et al. [113], scheme 2	185	—	20.34
Wen et al. [120], scheme 1	269	—	20.14
Kim et al. [69], scheme 2	272	43.40	20.10
chapter 4, scheme 1	240	42.83	19.93
chapter 4, scheme 2	151	42.69	19.83
ours, with $\lambda = 1.5 \times 10^{-5}$	179	41.64	19.22
ours, with $\lambda = 2.0 \times 10^{-5}$	156	42.44	19.65

Table 6.2: Our low-rank AlexNet models (with rank and scheme selection) and comparison to other low-rank results in the literature. We report top-1 and top-5 validation accuracy on ImageNet dataset and the FLOPs count of the final model.

pressed VGG16 nets do not experience accuracy drop until reaching models with 61 MFLOPs (×5.1 speed-up). In fact, for VGG16 we see a substantial improvement in test error for moderately compressed models: our 78 MFLOPs network has a test error of 5.97%, which is a 0.54% improvement wrt reference model. We also plot recent results from the structured pruning literature (as square markers) that reduce the FLOPs count of VGG16 [47, 48, 81, 85, 86, 136]. Our results *achieve significantly better error-FLOPs tradeoff* compared to low-rank compression using individual reshaping schemes and when compared to structured pruning results as well.

To illustrate the differences in selected ranks and schemes of our compressed models, we report some of the final architectures for VGG16 in Table 6.1. We notice non-trivial changes in both ranks and schemes of the final architectures: while a network with 156 MFLOPs has a mix of schemes 1, 2, and 3, the 78 MFLOPs network only uses schemes 1 and 2.

For the AlexNet experiments, we report the achieved FLOPs count and top-1/top-5 validation errors in Table 6.2. Our rank-and-scheme-optimized AlexNet models achieve better error-FLOPs tradeoff than most of the low-rank compression results existing in the literature and comparable to rank-optimized AlexNets of [53] which use scheme 2 throughout the network. Interestingly, our algorithm selects the scheme-2 decomposition for all convolutional layers of AlexNet as well, suggesting that scheme 2 might be a good default option for a high-compression regime.

Chapter 7

LC toolkit: open-source compression framework

An overarching theme of this dissertation is the low-rank compression of neural networks and its solution using learning-compression algorithm. As we have presented various extensions of this compression mechanism in chapters 4–6, the reader might have noticed that the only difference between the resulting algorithms in each case was the solution of the C-step problem. This is not a coincidence, but a result of the application of learning-compression algorithm for the model compression problems defined in a constrained formulation (see eq. 3.1).

Over the years, we have used the LC algorithm to train models using different compression mechanisms: quantization [16, 58], pruning [17, 58], low-rank compression [53, 55, 56, 57], and various combinations of those [55, 58, 59]. For the purposes of these experiments, we have created a software frameworks that capitalizes on the advantages of the LC algorithm. Now, after many rounds of internal testing, refactoring, and rewriting, we are releasing this code as an open-source library available for general audience.

7.1 LC algorithm: a software perspective

Although we have discussed the LC algorithm in great detail over the course of this dissertation, our previous encounters were primarily of academic nature.

The pseudocode of the LC algorithm

```
input training data and model with parameters $\mathbf{w}$
$\mathbf{w} \leftarrow \overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$                      pretrained model
$\mathbf{\Theta} \leftarrow \mathbf{\Theta}^{\mathrm{DC}} = \mathbf{\Pi}(\overline{\mathbf{w}})$                      init compression
$\boldsymbol{\beta} \leftarrow \mathbf{0}$
for $\mu = \mu_0 < \mu_1 < \cdots < \infty$
  $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta}) - \frac{1}{\mu}\boldsymbol{\beta}\|^2$          L step
  $\mathbf{\Theta} \leftarrow \arg\min_{\mathbf{\Theta}} \|\mathbf{w} - \frac{1}{\mu}\boldsymbol{\beta} - \mathbf{\Delta}(\mathbf{\Theta})\|^2 + \lambda\, C(\mathbf{\Theta})$          C step
  $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \mu(\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta}))$                      multipliers step
  if $\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|$ is small enough then exit the loop
return $\mathbf{w}$, $\mathbf{\Theta}$
```

Implementation of the LC algorithm in our software

```
class LCAlgorithm():
  # Housekeeping code ...
  # Pretrained model is provided by user at initialization
  def run(self):
    self.mu = 0
    self.c_step(step_number=0)
    for step_n, mu in enumerate(self.mu_schedule):
      self.mu = mu
      self.l_step(step_n) # call to user-provided L step
      self.c_step(step_n) # resolve the compression tasks
      self.multipliers_step()
```

Figure 7.1: The pseudocode of the LC algorithm using the augmented Lagrangian formulation and corresponding implementation in our software (located in `LCAlgorithm` class); the main running method is shown.

Type	Forms
Quantization	Adaptive Quantization into $\{c_1, c_2, \dots, c_K\}$ Binarization into $\{-1, 1\}$ and $\{-c, c\}$ Ternarization into $\{-c, 0, c\}$
Pruning	ℓ_0-constraint (s.t., $\|\mathbf{w}\|_0 \leq \kappa$) ℓ_1-constraint (s.t., $\|\mathbf{w}\|_1 \leq \kappa$) ℓ_0-penalty ($\alpha\|\mathbf{w}\|_0$) ℓ_1-penalty ($\alpha\|\mathbf{w}\|_1$)
Low-rank	Low-rank compression to a given rank Low-rank with *automatic* rank selection for FLOPs Low-rank with *automatic* rank selection for storage
Additive Combinations	Quantization + Pruning Quantization + Low-rank Pruning + Low-rank Quantization + Pruning + Low-rank

Table 7.1: Currently supported compression types, with their exact forms. These compression can be defined per one or multiple layers, and different compression can be applied to different parts of the model.

In this section, let us take a look at the algorithm from the software engineering perspective. To make the discussion easier we duplicate the pseudocode of the LC algorithm along with the actual implementation in Figure 7.1.

One of the main advantages of the LC algorithm is the separation of the model learning from the model compression which is encapsulated in alternation of these two (L and C) steps in the optimizaiton. Our software capitalizes on this separation: to apply a new compression mechanism under the LC formulation, the software requires only a new C step corresponding to the chosen compression mechanism, and the L step will be simply reused. Indeed, the compression parameter $\mathbf{\Theta}$ enters the L step problem as a constant regardless of the chosen compression type. Therefore, all L steps for any combination of compressions have the same form. Once the L step has been implemented for a model, any possible compression (C steps) can be applied.

Importantly, the separation of L and C steps allows us to use the best tools available for each steps. For modern neural networks, the L step optimization means performing iterations over the dataset (using SGD) and requires hardware accelerators. The formulation of the C step, on the other hand, is given by ℓ_2 minimization, and as we have seen in this dissertation, solutions of it can be computed using efficient algorithms. The list of currently supported compressions is given in Table 7.1.

Finally, from the software engineering perspective, the separation of L and C steps makes code robust and allows us to thoroughly test and debug each component separately. Other advantages of this separation include the following:

- **Modularity**. Each L and C step is implemented as a separate module, hence, changing the model or the compression type simply involves calling the corresponding routine.
- **Extensibility**. New machine learning models or compression technieques can be easily added by creating new modules.
- **Reusability**. Reusability happens on multiple levels. Compressions, and their C steps, will be reused across many models. The L step has to be implemented only once for a model, and it will be reused across multiple compressions. Additionally, many C step solutions can be directly lifted from specialized libraries that implement the necessary functionality (e.g., SVD) efficiently.
- **Usability**. In practice, one does not know what type of compression is best for a given model. Our approach offers, within the same framework of the LC algorithm, multiple models and multiple compression types that user can try or combine with minimum coding and engineering effort.

7.2 Design

Our main goal in designing the software is to have an easy to use, efficient, robust, and configurable neural network compression software. Particularly, we want to have the flexibility of applying any available compression (Table 7.1) to any parts of the neural network with per-layer granularity. For example, consider the following compression tasks:

- a single compression per layer: say, low-rank compression for layer 1 with target rank of 5
- a single compression per multiple layers: e.g., prune 5% of weights in layer 1 and 3, jointly
- mix compressions: e.g., quantize layer 1 and prune jointly layers 2 and 3
- additive compressions: be able to use additive compressions in the same mix-and-match way, for a single layer or multiple layers jointly

The mix-and-match on the level of a layer granularity is an important requirement as neural networks can have heterogeneous structures: having layers with few parameters but many FLOPs and vice-versa. As such, some layers might be better suited to the specific form of compression than others, which has been exploited in the literature with specific schemes targeting only, for example, fully-connected layers [18, 109]. To implement our desiderata, we leverage the modularity of the LC algorithm and introduce some additional building blocks next.

L step We hand off the model training operations, the L step, to the user through the lambda functions. This gives a fine-grained control to the user on the model's actual learning: hardware utilization, data source pulling, and other essential steps required for training. Usually, the L step implementation is already available or can be extracted from the training code used for the reference (uncompressed) model. Below we give a typical way of implementing the L step in PyTorch:

```
def my_l_step(model, lc_penalty, args**):
    loss = model.loss(out_, target_) + lc_penalty()
    loss.backward()
    optimizer.step()
```

Here we skipped some code (such as the setup of the optimizer and data source configuration) for brevity. Note that the only required change is the addition of `lc_penalty` term.

C step All provided compressions of Table 7.1 are implemented as subclasses of `CompressionTypeBase` class, and the actual C step is exposed through the `compress` method. This allows a straightforward extension of the library of compressions: if needed, the user simply wraps the custom C-step solution into an object of `CompressionTypeBase` class. Below we give an example implementation of the C step for binarization:

```
class ScaledBinaryQuantization(CompressionTypeBase):
    def compress(self, data):
        a = np.mean(np.abs(data))
        quantized = 2 * a * (data > 0) - a
        return quantized
```

Compression tasks To instruct the framework on which compression types should be applied to which parts of the model, the user needs to populate a compression tasks structure. This structure is a list of simple mappings of the form: (parameters) → (compression view, compression type), which is implemented as a python dictionary. The *parameters* are the subset of model weights, which are wrapped into internal `Parameter` object. The *compression view* is another internal structure that handles reshaping of the model weights into a form suitable for compression, e.g., reshaping the weight tensor of a convolutional layer into a matrix for low-rank compression.

While our strategy of defining the compression tasks might seem unnecessarily complicated, it brings *a considerable amount of flexibility.* For instance, it erases the limitations of standard compression approaches with coarse layer-based granularity: we can compress multiple layers with a single compression, or a single layer with multiple compressions, while simultaneously mixing different compressions in a single model. This abstraction disentangles compression from the model structure and allows us to construct complicated schemes of compressions in a *mix-and-match* way. For example, consider the following compression task:

$$\begin{aligned} &\text{(layer 1, layer 3)} && \rightarrow \text{(as a vector, adaptive quantization } k=6\text{)},\\ &\text{(layer 2)} && \rightarrow \text{(as is, low-rank with } r=3\text{)} \end{aligned}$$

where we want to jointly compress a three-layer neural network so that the first and third layers are quantized with the same codebook, and the second layer is

```
lc_alg = lc.Algorithm(
    model,                  # a model to compress
    compression_tasks,      # specifications of compression
    l_step_optimization,    # implementation of the L step
    mu_schedule,            # schedule of the mu values
    evaluation_func         # the evaluation function
)
lc_alg.run()                # an entry point to the LC algorithm
```

Figure 7.2: An example of running the LC algorithm in the toolkit.

a low-rank matrix with $r = 3$, and we want these compression to be applied simultaneously. The semantics of this compression is translated almost verbatim in our framework:

```
from lc.torch import ParameterTorch as P, AsVector, AsIs
compression_tasks = {
  P([l1.weight, l3.weight]): (AsVector, AdaptiveQuantization(k=6)),
  P(l2.weight):              (AsIs,     LowRank(target_rank=3))
}
```

The fine-grained control over semantics of the compression allows us to include expert knowledge about properties of a particular model (e.g., do not quantize the first layer) without much effort.

Running the software To compress a model, the user needs to construct an `lc.Algorithm` object and provide the following: 1) a model to be compressed 2) associated compression tasks 3) implementation of the L step 4) a schedule of μ values, and 5) an evaluation function to keep track of the error during the compression. We give an example of running the algorithm in Figure 7.2.

Once the `run` method is called, the LC algorithm will start execution, at which point the library will proceed in line-by-line correspondence to the pseudocode on the top of Figure 7.1. Currently, each of the compression tasks (and corresponding C step implementation) is called in order. Yet, due to the nature of the LC algorithm, every compression task's C steps can be executed in parallel, further improving the efficiency of the toolkit.

7.3 A guided tour through the functionality

In this section, we demonstrate the flexibility of our framework by easily exploring multiple compression schemes with minimal effort. As an example, say we are tasked with compressing the storage bits of the LeNet300 neural network trained on MNIST dataset (10 classes, 28 × 28 gray-scale images). The LeNet300 is a three-layer neural network with 300, 100, and 10 neurons respectively on every layer; the reference network has an error of 1.66% on the test set.

In order to run the LC algorithm, we need to provide an L step implementation and compression tasks as described in sec. 5. The implementation of corresponding L step is given in Figure 7.4. Now, having the L step implementation, we can formulate the compression tasks. Say, we would like to know what would be the test error if the model is optimally quantized with a separate codebook on each layer? Test error in such case is 1.97%, which is 0.31% higher than the reference. What would be the performance of the model if one would quantize only the first and the third layers, leaving the second layer untouched? Test error in such case is 1.96%. What about if we prune all but 5% of the weights? Yes, our framework can handle all of these combinations and more; see Figure 7.3 for other examples. We can even apply different compressions to every layer, for example, take a look at the example of Figure 7.3, where we apply quantization, pruning, and low-rank compression to the different parts of the LeNet300. Once the L step is given, trying a new compression scheme only requires a new compression task.

7.4 Practical advice

We implemented the LC algorithm originally in 2017, and have gone through multiple refinements and code reimplementations. We have applied it to compressing a wide array of relatively large neural nets, such as AlexNet, VGG, ResNet, etc., which are themselves tricky to train well in the first place. In the process, we have gathered a considerable amount of practical knowledge on the behavior of the LC algorithm on both small and large models and datasets. We want to share a list of common pitfalls so future users of our toolkit would hopefully avoid them.

Quantize all layers, test error: 1.97%

```
compression_tasks = {
  Param(l1.weight): (AsVector, AdaptiveQuantization(k=2)),
  Param(l2.weight): (AsVector, AdaptiveQuantization(k=2)),
  Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))
}
```

Quantize first and third layers, test error: 1.96%

```
compression_tasks = {
  Param(l1.weight): (AsVector, AdaptiveQuantization(k=2)),
  Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))
}
```

Prune all but 5%, test error: 1.70%

```
compression_tasks = {
  Param([l1.weight, l2.weight, l3.weights]):
    (AsVector, ConstraintL0Pruning(kappa=13310)) # 13310 = 5%
}
```

Single codebook quantization with 1% non-zeros, test error: 1.85%

```
compression_tasks = { Param([l1.weight, l2.weight, l3.weights]): [
    (AsVector, ConstraintL0Pruning(kappa=2662)), # 2662 = 1%
    (AsVector, AdaptiveQuantization(k=2))]
}
```

Prune first layer, low-rank to second, quantize third, test error: 1.68%

```
compression_tasks = {
  Param(l1.weight): (AsVector, ConstraintL0Pruning(kappa=5000)),
  Param(l2.weight): (AsIs,     LowRank(target_rank=10))
  Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))
}
```

Figure 7.3: Some of the mix-and-match compressions possible in our framework and corresponding train/test errors. Here, we use the LeNet300 neural network trained on the MNIST dataset (reference test error is 1.66%) and report final test errors after compression. Notice that trying a new combination of compressions is as simple as writing a new *compression tasks* structure.

```
def my_l_step(model, lc_penalty, step):
  params = [p for p in model.parameters() if p.requires_grad]
  lr = lr_base*(0.98**step)                 # decayed learning rate
  optimizer = optim.SGD(params, lr=lr, momentum=0.9, nesterov=True)
  for epoch in range(epochs_per_step):
    for x, target in train_loader:          # loop over the dataset
      optimizer.zero_grad()
      loss = model.loss(model(x), target) + lc_penalty()
      loss.backward()
      optimizer.step()
```

Figure 7.4: An example implementation of the L step for LeNet300.

- **Monitor the progression of the algorithm** Specifically, two important quantities to keep an eye on:
  - The loss of the L step: $L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$. The total loss at the end of the L step must be smaller than the total loss at the beginning. If some L step has not reduced the loss, optimization parameters of the step should be tuned.
  - The loss of the C step, $\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$, must have a smaller value after each C step. This often fails when new compression is introduced into the pipeline, where `compress` method is not fully tested. For the base compressions in the framework, we made sure they always optimize the C step.

- **On the μ schedule** Theoretically, the sequence of μ values should start at 0 and infinitesimally grow to ∞. In practice, we use an exponentially increasing schedule $\mu_k = \mu_0 \times a^k$ with small initial μ_0 and appropriately chosen $a > 1$ for the k-th step of the LC. For most of compression schemes, we have developed robust estimates of μ_0-values: for pruning see suppl.mat. of [17], for rank-selection see suppl.mat. of [53]. For the value of a, we found the range of [1.1 1.4] to be a good spot.

Chapter 8

Conclusion and future work

In this dissertation we studied the problem of neural network compression and considered a particular compression of interest: the low-rank compression and several extensions of it. We provided an efficient solution based on the learning-compression algorithm of Carreira-Perpiñán [15] and presented a software framework that implements the algorithm and all discussed compressions (and many others). The resulting optimization approach have the same algorithmic structure and alternate two simple steps: L step that learns the model and C step that compresses the model according to chosen compression. In terms of actual compression results, the presented algorithms are competitive and have been published in peer reviewed conferences [53, 56, 57, 59]. We also created an extensible neural network compression software that supports all discussed compression schemes (and many others) and allows us to compress any neural network model with minimal effort [52, 54].

Due to the decoupling of model training (L step) from compression (C step) provided by the LC algorithm there are many possible extensions to our work. Decoupling allows us to explore different compression schemes; and while we have presented only low-rank based applications in this paper, we have extended our research to include quantization of model weights [16], pruning [17] and additive combinations of several compressions [55, 59]. In the remaining part of this section we discuss several future projects which can be built upon our work.

Selection mechanisms One possible future direction is to further develop the idea of selection given by $\lambda C(\boldsymbol{\Theta})$ term in eq. (3.1), which we have used for the selection of the best rank/shape of a low-rank compression (chapters 4 and 6). This can be achieved by developing new cost functions, including:

- cost function on power consuption, so that compressed model will use less energy or will be operating at lower temperatures,
- cost function on model/hardware layout and placement which would allow us to, for instance, determine whether we need to run the parts of the compressed model on GPU or CPU,
- cost function on other metrics like utilization so the model will have fewer number of cache misses, higher transfer speeds and loading times.

Compression of other ML models So far we have been studying the problems of neural network compression, which are warranted by large size of the models and huge demand to shrink them for various industrial applications. However, compression of other (non-neural-network) machine learning models pose an intersting research question, and has not been studied to a larger extent in the literature. Only some of the ML models have been considered in the context of particular compressions and have efficient algorithmic solutions:

- in context of pruning, sparse linear models are well known, with famous LASSO formulation for regression and extensions to support vector machines
- in context of low-rank compression, the reduced rank regression (RRR) optimally solves the problem for linear regression.

If you want to compress other models or would like to consider other compression (say quantization, or combination of pruning and quantization), you need to look for a new algorithm. The decoupling of L and C steps of the LC algorithm is well suited for compression of any model as long as its learning can be efficiently captured in the L step, thus we can further explore the application of the LC algorithms for ML models. Some models of interest would be Gaussian Mixture Models and Hidden Markov Models.

Appendix A

Number of floating point operations, FLOPs

There is no clear consensus in the literature on how to compute floating point operations in the forward pass of a neural network. While some authors define this number as total number of multiplications and additions [128], others count one multiplication and addition as one operation [45], assuming multiplication and addition will be fused during the forward pass. When reporting the FLOPs count in this work we stick to the latter definition of the FLOPS – total number of fused multiplications and additions incurred by convolutional or fully connected layers.

For example, assume we have a layer with weights $\mathbf{W} \in \mathbb{R}^{300\times 784}$ and biases $\mathbf{b} \in \mathbb{R}^{300}$. According to the first definition, the total number of FLOPs of $\mathbf{Wx}+\mathbf{b}$ for input $\mathbf{x} \in \mathbb{R}^{784}$ is

$$\text{FLOPs} = \underbrace{784 \times 300}_{\text{multiplications in } \mathbf{Wx}} + \underbrace{783 \times 300}_{\text{additions in } \mathbf{Wx}} + \underbrace{300}_{\text{for adding } \mathbf{b}} = 470400.$$

However, according to second definition, this number will be only $784 \times 300 = 235200$ as multiplications and additions are fused.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, Ł. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. TensorFlow WhitePaper.

[2] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, May 2009.

[3] J. M. Alvarez and M. Salzmann. Learning the number of neurons in deep networks. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 2270–2278. MIT Press, Cambridge, MA, 2016.

[4] J. M. Alvarez and M. Salzmann. Compression-aware training of deep networks. In I. Guyon, U. v. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 30, pages 856–867. MIT Press, Cambridge, MA, 2017.

[5] R. Alvarez, R. Prabhavalkar, and A. Bakhtin. On the efficient representation and execution of deep acoustic models. In *Proc. of Interspeech'16*, pages 2746–2750, San Francisco, CA, Sept. 8–12 2016.

[6] S. Anwar, K. Hwang, and W. Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'15)*, pages 1131–1135, Brisbane, Australia, Apr. 19–24 2015.

[7] R. Banner, Y. Nahshan, and D. Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NEURIPS)*, volume 32, pages 7950–7958. MIT Press, Cambridge, MA, 2019.

[8] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Imaging Sciences*, 2(1):183–202, 2009.

[9] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

[10] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfigurable Technology and Systems*, 11(3):16.1–16.23, Dec. 2018.

[11] J. D. Bruce. Optimum quantization. Technical Report 429, Massachussetts Institute of Technology, 1965.

[12] J.-F. Cai, E. J. Candès, and Z. Shen. A singular value thresholding algorithm for matrix completion. *SIAM J. Optimization*, 20(4):1956–1982, 2010.

[13] Z. Cai, X. He, J. Sun, and N. Vasconcelos. Deep learning with low precision by half-wave Gaussian quantization. In *Proc. of the 2017 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'17)*, pages 5918–5926, Honolulu, HI, July 21–26 2017.

[14] E. J. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772, Dec. 2009.

[15] M. Á. Carreira-Perpiñán. Model compression as constrained optimization, with application to neural nets. Part I: General framework. arXiv:1707.01209, July 5 2017.

[16] M. Á. Carreira-Perpiñán and Y. Idelbayev. Model compression as constrained optimization, with application to neural nets. Part II: Quantization. arXiv:1707.04319, July 13 2017.

[17] M. Á. Carreira-Perpiñán and Y. Idelbayev. "Learning-compression" algorithms for neural net pruning. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 8532–8541, Salt Lake City, UT, June 18–22 2018.

[18] P. Chen, S. Si, Y. Li, C. Chelba, and C.-J. Hsieh. GroupReduce: Block-wise low-rank approximation for neural language model shrinking. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NEURIPS)*, volume 31, pages 10988–10998. MIT Press, Cambridge, MA, 2018.

[19] J. Choi, S. Venkataramani, V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang. Accurate and efficient 2-bit quantized neural networks. In *Proc. of the 2nd Conf. Systems and Machine Learning (SysML 2019)*, Stanford, CA, Mar. 31 – Apr. 2 2019.

[20] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proc. of the 2017 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'17)*, pages 1800–1807, Honolulu, HI, July 21–26 2017.

[21] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev. Low-bit quantization of neural networks for efficient inference. In *ICCV Workshop on Compact and Efficient Feature Representation and Learning in Computer Vision*, 2019.

[22] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. In C. Cortes,

N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 3105–3113. MIT Press, Cambridge, MA, 2015.

[23] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1. arXiv:1602.02830, Mar. 17 2016.

[24] S. Dasgupta and Y. Freund. Random projection trees for vector quantization. *IEEE Trans. Information Theory*, 55(7):3229–3242, July 2009.

[25] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 26, pages 2148–2156. MIT Press, Cambridge, MA, 2013.

[26] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 27, pages 1269–1277. MIT Press, Cambridge, MA, 2014.

[27] C. Ding, D. Zhou, X. He, and H. Zha. R_1-PCA: Rotational invariant l_1-norm principal component analysis for robust subspace factorization. In W. W. Cohen and A. Moore, editors, *Proc. of the 23rd Int. Conf. Machine Learning (ICML'06)*, pages 281–288, Pittsburgh, PA, June 25–29 2006.

[28] X. Ding, G. Ding, Y. Guo, J. Han, and C. Yan. Approximated oracle filter pruning for destructive CNN width optimization. In K. Chaudhuri and R. Salakhutdinov, editors, *Proc. of the 36th Int. Conf. Machine Learning (ICML 2019)*, pages 1607–1616, Long Beach, CA, June 9–15 2019.

[29] X. Dong, S. Chen, and S. Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In I. Guyon, U. v. Luxburg, S. Bengio,

H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 30, pages 4857–4867. MIT Press, Cambridge, MA, 2017.

[30] J. Fang, A. Shafiee, H. Abdel-Aziz, D. Thorsley, G. Georgiadis, and J. Hassoun. Post-training piecewise linear quantization for deep neural networks. arXiv:2002.00104, Jan. 31 2020.

[31] M. Fazel. *Matrix Rank Minimization with Applications*. PhD thesis, Stanford University, Mar. 2002.

[32] E. Fiesler, A. Choudry, and H. J. Caulfield. Weight discretization paradigm for optical neural networks. In *Proc. SPIE 1281: Optical Interconnections and Networks*, pages 164–173, The Hague, Netherlands, Aug. 1 1990.

[33] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov. Ultimate tensorization: Compressing convolutional and FC layers alike. arXiv:1611.03214, Nov. 10 2016.

[34] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proc. of the 13th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2010)*, pages 249–256, Chia Laguna, Sardinia, Italy, Mar. 21–24 2010.

[35] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, fourth edition, 2012.

[36] J. Gusak, M. Kholiavchenko, E. Ponomarev, L. Markeeva, P. Blagoveschensky, A. Cichocki, and I. Oseledets. Automated multi-stage compression of neural networks. In *ICCV Workshop on Low-Power Computer Vision*, 2019.

[37] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 1135–1143. MIT Press, Cambridge, MA, 2015.

[38] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *Proc. 43rd Int. Symposium on Computer Architecture (ISCA 2016)*, pages 243–254, Seoul, Korea, June 18–22 2016.

[39] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[40] S. J. Hanson and L. Y. Pratt. Comparing biases for minimal network construction with back-propagation. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS)*, volume 1, pages 177–185. Morgan Kaufmann, San Mateo, CA, 1989.

[41] Z. Harchaoui, M. Douze, M. Paulin, M. Dudik, and J. Malick. Large-scale image classification with trace-norm regularization. In *Proc. of the 2012 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'12)*, pages 3386–3393, Providence, RI, June 16–21 2012.

[42] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 5, pages 164–171. Morgan Kaufmann, San Mateo, CA, 1993.

[43] T. Hastie, R. Tibshirani, and M. Wainwright. *Statistical Learning with Sparsity: The Lasso and Generalizations.* Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, 2015.

[44] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proc. 15th Int. Conf. Computer Vision (ICCV'15)*, pages 1026–1034, Santiago, Chile, Dec. 11–18 2015.

[45] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of the 2016 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'16)*, pages 770–778, Las Vegas, NV, June 26 – July 1 2016.

[46] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proc. 16th Int. Conf. Computer Vision (ICCV'17)*, pages 1398–1406, Venice, Italy, Dec. 11–18 2017.

[47] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19)*, pages 4340–4349, Long Beach, CA, June 16–20 2019.

[48] Z. Huang and N. Wang. Data-driven sparse structure selection for deep neural networks. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Proc. 15th European Conf. Computer Vision (ECCV'18)*, pages 304–320, Munich, Germany, Sept. 8–14 2018.

[49] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and −1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Belfast, UK, Oct. 20–22 2014.

[50] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50*times* fewer parameters and <0.5MB model size. arXiv:1602.07360 [cs.CV], Nov. 4 2016.

[51] Y. Idelbayev. Proper ResNet implementation for CIFAR10/CIFAR100 in PyTorch. `https://github.com/akamaster/pytorch_resnet_cifar10`. Accessed: 2020-May-15.

[52] Y. Idelbayev and M. Á. Carreira-Perpiñán. LC: A flexible, extensible open-source toolkit for model compression. In *Conference on Information and Knowledge Management (CIKM 2021).*

[53] Y. Idelbayev and M. Á. Carreira-Perpiñán. Low-rank compression of neural nets: Learning the rank of each layer. In *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'20)*, pages 8046–8056, Seattle, WA, June 14–19 2020.

[54] Y. Idelbayev and M. Á. Carreira-Perpiñán. A flexible, extensible software framework for model compression based on the LC algorithm. arXiv:2005.07786, May 15 2020.

[55] Y. Idelbayev and M. Á. Carreira-Perpiñán. Neural network compression via additive combination of reshaped, low-rank matrices. In A. Bilgin, M. W. Marcellin, J. Serra-Sagrista, and J. A. Storer, editors, *Proc. Data Compression Conference (DCC 2021)*, pages 243–252, Online, Mar. 23–26 2021.

[56] Y. Idelbayev and M. Á. Carreira-Perpiñán. Optimal selection of matrix shape and decomposition scheme for neural network compression. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'21)*, pages 3250–3254, Toronto, Canada, June 6–11 2021.

[57] Y. Idelbayev and M. Á. Carreira-Perpiñán. Beyond FLOPs in low-rank compression of neural networks: Optimizing device-specific inference runtime. In *IEEE Int. Conf. Image Processing (ICIP 2021)*, Anchorage, AK, Sept. 19–22 2021.

[58] Y. Idelbayev and M. Á. Carreira-Perpiñán. An empirical comparison of quantization, pruning and low-rank neural network compression using the LC toolkit. In *Int. J. Conf. Neural Networks (IJCNN'21)*, Virtual event, July 18–22 2021.

[59] Y. Idelbayev and M. Á. Carreira-Perpiñán. More general and effective model compression via an additive combination of compressions. In *Proc. of the 32nd European Conf. Machine Learning (ECML–21)*, Bilbao, Spain, Sept. 13–17 2021.

[60] Y. Idelbayev, P. Molchanov, M. Shen, H. Yin, M. Á. Carreira-Perpiñán, and J. M. Alvarez. Optimal quantization using scaled codebook. In *Proc. of the 2021 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'21)*, pages 12095–12104, Virtual, June 19–25 2021.

[61] Y. Ioannou, D. Robertson, J. Shotton, R. Cipolla, and A. Criminisi. Training CNNs with low-rank filters for efficient image classification. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[62] Y. Ioannou, D. Robertson, R. Cipolla, and A. Criminisi;. Deep Roots: Improving CNN efficiency with hierarchical filter groups. In *Proc. of the 2017 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'17)*, pages 1231–1240, Honolulu, HI, July 21–26 2017.

[63] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In F. Bach and D. Blei, editors, *Proc. of the 32nd Int. Conf. Machine Learning (ICML 2015)*, pages 448–456, Lille, France, July 6–11 2015.

[64] A. J. Izenman. Reduced-rank regression for the multivariate linear model. *J. Multivariate Analysis*, 5(2):248–264, June 1975.

[65] 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 *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 2704–2713, Salt Lake City, UT, June 18–22 2018.

[66] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In M. Valstar, A. French, and T. Pridmore, editors, *Proc. of the 25th British Machine Vision Conference (BMVC 2014)*, Nottingham, UK, Sept. 1–5 2014.

[67] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. arXiv:1408.5093 [cs.CV], June 20 2014.

[68] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proc. 44th Int. Symposium on Computer Architecture (ISCA 2017)*, pages 1–12, Toronto, Canada, June 24–28 2017.

[69] H. Kim, M. U. K. Khan, and C.-M. Kyung. Efficient neural network compression. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19)*, pages 12569–12577, Long Beach, CA, June 16–20 2019.

[70] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[71] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[72] J. Kossaifi, A. Toisoul, A. Bulat, Y. Panagakis, T. Hospedales, and M. Pantic. Factorized higher-order CNNs with an application to spatio-temporal

emotion estimation. In *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'20)*, pages 6060–6069, Seattle, WA, June 14–19 2020.

[73] A. Kozlov, I. Lazarevich, V. Shamporov, N. Lyalyushkin, and Y. Gorbachev. Neural network compression framework for fast model inference. arXiv:2002.08679, Feb. 20 2020.

[74] R. Krishnamoorth. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv:1806.08342, June 21 2018.

[75] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 25, pages 1106–1114. MIT Press, Cambridge, MA, 2012.

[76] V. Lebedev and V. Lempitsky. Fast ConvNets using group-wise brain damage. In *Proc. of the 2016 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'16)*, pages 2554–2564, Las Vegas, NV, June 26 – July 1 2016.

[77] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *Proc. of the 3rd Int. Conf. Learning Representations (ICLR 2015)*, San Diego, CA, May 7–9 2015.

[78] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS)*, volume 2, pages 598–605. Morgan Kaufmann, San Mateo, CA, 1990.

[79] C. Leng, H. Li, S. Zhu, and R. Jin. Extremely low bit neural network: Squeeze the last bit out with ADMM. In *Proc. of the 32nd AAAI Conference*

on Artificial Intelligence (AAAI 2018), pages 3466–3473, New Orleans, LA, Feb. 2–7 2018.

[80] C. Li and C. J. R. Shi. Constrained optimization based low-rank approximation of deep neural networks. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Proc. 15th European Conf. Computer Vision (ECCV'18)*, pages 746–761, Munich, Germany, Sept. 8–14 2018.

[81] H. Li, A. Kadav, I. Durdanovic, and H. P. Graf. Pruning filters for efficient ConvNets. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017)*, Toulon, France, Apr. 24–26 2017.

[82] J. Li, Q. Qi, J. Wang, C. Ge, Y. Li, Z. Yue, and H. Sun. OICSR: Out-in-channel sparsity regularization for compact deep neural networks. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19)*, pages 7046–7055, Long Beach, CA, June 16–20 2019.

[83] D. Lin, S. Talathi, and S. Annapureddy. Fixed point quantization of deep convolutional networks. In M.-F. Balcan and K. Q. Weinberger, editors, *Proc. of the 33rd Int. Conf. Machine Learning (ICML 2016)*, pages 2849–2858, New York, NY, June 19–24 2016.

[84] M. Lin, Q. Chen, and S. Yan. Network in network. In *Int. Conf. Learning Representations (ICLR 2014)*, 2014.

[85] M. Lin, R. Ji, Y. Wang, Y. Zhang, B. Zhang, Y. Tian, and L. Shao. HRank: Filter pruning using high-rank feature map. In *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'20)*, pages 1526–1535, Seattle, WA, June 14–19 2020.

[86] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann. Towards optimal structured CNN pruning via generative adversarial learning. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19)*, pages 2790–2799, Long Beach, CA, June 16–20 2019.

[87] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common objects in context. In *Proc. 13th European Conf. Computer Vision (ECCV'14)*, pages 740–755, Zürich, Switzerland, Sept. 6–12 2014.

[88] B. Liu, M. Wan, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'15)*, pages 806–814, Boston, MA, June 7–12 2015.

[89] X. Liu, M. Ye, D. Zhou, and Q. Liu. Post-training quantization with multiple points: Mixed precision without mixed precision. arXiv:2002.09049, Feb. 20 2020.

[90] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *Proc. 16th Int. Conf. Computer Vision (ICCV'17)*, Venice, Italy, Dec. 11–18 2017.

[91] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Information Theory*, 28(2):129–137, Mar. 1982.

[92] D. M. Loroch, N. Wehn, F.-J. Pfreundt, and J. Keuper. TensorQuant — a simulation toolbox for deep neural network quantization. In *Proc. SC Workshop on Machine Learning on HPC Environments (MLHPC'17)*, pages 1:1–1:8, Nov. 2017.

[93] J. MacQueen. Some methods for classication and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proc. Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[94] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. The planar k-means problem is NP-hard. *Theoretical Computer Science*, 442:13–21, July 13 2012.

[95] M. B. Milde, D. Neil, A. Aimar, T. Delbruck, and G. Indiveri. ADaPTION: Toolbox and benchmark for training convolutional neural networks with re-

duced numerical precision weights and activation. arXiv:1711.04713, Nov. 13 2017.

[96] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017)*, Toulon, France, Apr. 24–26 2017.

[97] M. C. Mozer and P. Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS)*, volume 1, pages 107–115. Morgan Kaufmann, San Mateo, CA, 1989.

[98] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling. Data-free quantization through weight equalization and bias correction. In *Proc. 18th Int. Conf. Computer Vision (ICCV'19)*, pages 1325–1334, Seoul, Korea, Oct. 27 – Nov. 2 2019.

[99] S. Nakajima, M. Sugiyama, S. D. Babacan, and R. Tomioka. Global analytic solution of fully-observed variational Bayesian matrix factorization. *J. Machine Learning Research*, 14:1–37, Jan. 2013.

[100] Y. Nesterov. A method of solving a convex programming problem with convergence rate $\mathcal{O}(1/k^2)$. *Soviet Math. Dokl.*, 27(2):372–376, 1983.

[101] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

[102] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov. Tensorizing neural networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 442–450. MIT Press, Cambridge, MA, 2015.

[103] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in Py-

Torch. In *NIPS Workshop on The Future of Gradient-Based Machine Learning Software (Autodiff)*, 2017.

[104] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-net: ImageNet classification using binary convolutional neural networks. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Proc. 14th European Conf. Computer Vision (ECCV'16)*, pages 525–542, Amsterdam, The Netherlands, Oct. 11–14 2016.

[105] B. Recht, M. Fazel, and P. A. Parrilo. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. *SIAM Review*, 52 (3):471–501, Aug. 2010.

[106] G. C. Reinsel and R. P. Velu. *Multivariate Reduced-Rank Regression. Theory and Applications*. Number 136 in Lecture Notes in Statistics. Springer-Verlag, 1998.

[107] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[108] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *Int. J. Computer Vision*, 115(3): 211–252, Dec. 2015.

[109] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arısoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'13)*, pages 6655–6659, Vancouver, Canada, Mar. 26–30 2013.

[110] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 4510–4520, Salt Lake City, UT, June 18–22 2018.

[111] S. Shin, Y. Boo, and W. Sung. Fixed-point optimization of deep neural networks with adaptive step size retraining. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'17)*, pages 1203–1207, New Orleans, LA, Mar. 5–9 2017.

[112] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'15)*, pages 1–9, Boston, MA, June 7–12 2015.

[113] C. Tai, T. Xiao, Y. Zhang, X. Wang, and W. E. Convolutional neural networks with low-rank regularization. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[114] C. Z. Tang and H. K. Kwan. Multilayer feedforward neural networks with single powers-of-two weights. *IEEE Trans. Signal Processing*, 41(8):2724–2727, Aug. 1993.

[115] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. arXiv:1605.02688, May 9 2016.

[116] F. Tung and G. Mori. CLIP-Q: Deep network compression learning by in-parallel pruning-quantization. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 7873–7882, Salt Lake City, UT, June 18–22 2018.

[117] W. Wang, Y. Sun, B. Eriksson, W. Wang, and V. Aggarwal. Wide compression: Tensor ring nets. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 9329–9338, Salt Lake City, UT, June 18–22 2018.

[118] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman. Generalization by weight-elimination with application to forecasting. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information*

Processing Systems (NIPS), volume 3, pages 875–882. Morgan Kaufmann, San Mateo, CA, 1991.

[119] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 2074–2082. MIT Press, Cambridge, MA, 2016.

[120] W. Wen, C. Xu, C. Wu, Y. Wang, Y. Chen, and H. Li. Coordinating filters for faster deep neural networks. In *Proc. 16th Int. Conf. Computer Vision (ICCV'17)*, Venice, Italy, Dec. 11–18 2017.

[121] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. arXiv:2004.09602, Apr. 20 2020.

[122] J. Wu, Y. Zhang, H. Bai, H. Zhong, J. Hou, W. Liu, W. Huang, and J. Huang. PocketFlow: An automated framework for compressing and accelerating deep neural networks. In *NIPS Workshop on Compact Deep Neural Network Representation with Industrial Applications (CDNNRIA)*, 2018.

[123] X. Wu. Optimal quantization by matrix searching. *J. Algorithms*, 12(4): 663–673, Dec. 1991.

[124] X. Wu and J. Rokne. An $\mathcal{O}(KN \log N)$ algorithm for optimum K-level quantization on histograms of n points. In *Proc. 17th ACM Annual Computer Science Conference*, pages 339–343, Louisville, KY, Feb. 21–23 1989.

[125] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1:216–222, Apr. 17 2018.

[126] J. Xue, J. Li, and Y. Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In F. Bimbot, C. Cerisara,

C. Fougeron, G. Gravier, L. Lamel, F. Pellegrino, and P. Perrier, editors, *Proc. of Interspeech'13*, pages 2365–2369, Lyon, France, Aug. 25–29 2013.

[127] H. Yang, W. Wen, and H. Li. DeepHoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures. In *Proc. of the 8th Int. Conf. Learning Representations (ICLR 2020)*, Addis Ababa, Ethiopia, Apr. 26–30 2020.

[128] J. Ye, X. Lu, Z. Lin, and J. Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *Proc. of the 6th Int. Conf. Learning Representations (ICLR 2018)*, Vancouver, Canada, Apr. 30 – May 3 2018.

[129] J. Ye, L. Wang, G. Li, D. Chen, S. Zhe, X. Chu, and Z. Xu. Learning compact recurrent neural networks with Block-Term tensor decomposition. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pages 9378–9387, Salt Lake City, UT, June 18–22 2018.

[130] P. Yin, S. Zhang, Y. Qi, and J. Xin. Quantization and training of low bit-width convolutional neural networks for object detection. arXiv:1612.06052, Aug. 17 2017.

[131] P. Yin, S. Zhang, Y. Qi, and J. Xin. Training ternary neural networks with exact proximal operator. arXiv:1612.06052, Aug. 17 2017.

[132] D. Yu, F. Seide, G. Li, and L. Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'12)*, pages 4409–4412, Kyoto, Japan, Mar. 25–30 2012.

[133] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis. NISP: Pruning networks using neuron importance score propagation. In *Proc. of the 2018 IEEE Computer Society Conf. Com-*

puter Vision and Pattern Recognition (CVPR'18), pages 9194–9203, Salt Lake City, UT, June 18–22 2018.

[134] D. Zhang, J. Yang, D. Ye, and G. Hua. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Proc. 15th European Conf. Computer Vision (ECCV'18)*, pages 365–382, Munich, Germany, Sept. 8–14 2018.

[135] X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 38(10):1943–1955, Oct. 2016.

[136] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian. Variational convolutional neural network pruning. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19)*, pages 2780–2789, Long Beach, CA, June 16–20 2019.

[137] Y. Zhao, X. Gao, R. Mullins, and C. Xu. Mayo: A framework for auto-generating hardware friendly deep neural networks. In *Proc. 2nd Int. Workshop on Embedded and Mobile Deep Learning (EMDL'18)*, pages 25–30, Munich, Germany, June 2018.

[138] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv:1606.06160, July 17 2016.

[139] 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 S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NEURIPS)*, volume 31, pages 815–886. MIT Press, Cambridge, MA, 2018.

[140] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, May 1977.

[141] N. Zmora, G. Jacob, L. Zlotnik, B. Elharar, and G. Novik. Neural network distiller: a Python package for DNN compression research. arXiv:1910.12232, Oct. 27 2019.