Compact and efficient representations of deep neural networks

vorgelegt von
M. Sc.
Simon Wiedemann
ORCID: 0000-0001-5144-3758

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Klaus Obermayer
Gutachter: Prof. Dr. Klaus-Robert Müller
Gutachter: Prof. Dr.-Ing. Peter Eisert
Gutachter: Prof. Dr.-Ing. Benno Stabernack

Tag der wissenschaftlichen Aussprache: 7. Oktober 2021

Berlin 2022
Compact and efficient representations of deep neural networks

Author: Simon Wiedemann
Supervisor: Prof. Dr. Klaus-Robert Müller
Committee members: Prof. Dr.-Ing. Peter Eisert, Prof. Dr.-Ing. Benno Stabernack, Prof. Dr. Klaus Obermayer

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Engineering in the

Machine Learning Institute of Software Engineering and Theoretical Computer Science

Day of oral defense: October 07, 2021
Declaration of Authorship

I, Simon Wiedemann, declare that this thesis titled, “Compact and efficient representations of deep neural networks” and the work presented in it are my own. I confirm that:

- Any part of this thesis that has previously been submitted to any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:
IEEE Disclaimer

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of TU Berlins’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
“The first principle is that you must not fool yourself and you are the easiest person to fool.”

Richard Feynmann
Abstract

Faculty IV: Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science

Doctor of Engineering

Compact and efficient representations of deep neural networks
by Simon Wiedemann

The past decade has experienced a blistering progress in artificial intelligence, achieving several breakthroughs in a wide range of tasks such as image classification, voice and object recognition, translation, media generation, etc.; to even mastering the game of Go at superhuman levels. Most of the successes were achieved thanks to deep neural networks (DNNs), a particular class of machine learning algorithms that are able to capture complex patterns from large amounts of data. However, DNNs suffer from the caveat that they need to be equipped with vast amounts of parameters in order to learn meaningful representations from them, so much so that, in practice, most state-of-the-art models have hundreds of millions of parameters, and can sometimes even reach the billions. This renders DNN models to be very resource-hungry, requiring high amounts of memory, compute power and expensive hardware components in order to be executed. Consequently, deploying DNNs becomes unfeasible for a wide set of real-world use-cases, specially if inference on resource-constraint devices such as mobile phones or micro-controllers is required, thus hindering their wide adoption in the market.

Motivated by this issue, in this thesis we study methods that aim to reduce the memory and processing complexity of DNN models. In particular, our contributions mainly focus on reducing their memory requirements, since communication of information incurs highest costs (in terms of speed, power and economical costs) among all types of operations. Our approach consists of firstly characterizing the information content entailed in the models parameters, and then devise methods for minimizing it. We put particular focus on the design of compression algorithms that output highly compact representations of them. To this end, we propose two types of algorithms with different characteristics:

- **DeepCABAC**, outputs maximally compact representations but does not trivially reduce the complexity for performing inference,

- **CER & CSER**, output compact and computationally efficient representations simultaneously.
Finally, we devise a specialized hardware architecture, named FantastIC4, that is able to leverage on compressed representations in order to attain a highly efficient execution engine of DNNs.

Our experimental results show that our compression techniques achieve up to $\times 51$ size reduction on average, and that we can reduce the number of required operations for inference by $\times 5.53$. Moreover, with our hardware architecture we were able to increase the area efficiency (GOPS/mm$^2$) for performing inference by up to $\times 115$, two orders of magnitude better as compared to previous approaches.

Zusammenfassung


Motiviert durch dieses Problem, untersuchen wir in dieser Dissertation Methoden, die darauf abzielen, den Speicherbedarf und die Verarbeitungskomplexität von DNN Modellen zu reduzieren. Insbesondere konzentrieren sich unsere Beiträge auf die Verringerung von Speicherbedarf, da die Datenkommunikation unter allen Arten von Operationen die höchsten “Kosten” verursacht (Geschwindigkeit, Energie und wirtschaftliche Kosten). Unser Ansatz besteht darin zunächst den Informationsgehalt, der in den Modellparametern steckt, zu charakterisieren und anschließend Methoden zu entwickeln, die diese minimieren. Wir legen besonderen Fokus auf den Entwurf von Kompressionsalgorithmen, die hochkompakte Repräsentationen dieser Daten ausgeben. Wir schlagen zwei Methoden mit jeweils unterschiedlichen Eigenschaften vor:

- **DeepCABAC**: Besteht aus einem Kompressionsalgorithmus, der maximal kompakte Repräsentationen der DNN-Parameter ausgibt. Dadurch wird der Speicherbedarf drastisch reduziert, jedoch die Rechenkomplexität der Inferenz durch den Algorithmus vernachlässigt.
• **CER & CSER**: Besteh aus zwei Kompressionsalgorithmen, die hoch kompakte Repräsentationen ausgeben und gleichzeitig die Rechenkomplexität reduzieren.

Schließlich entwickeln wir eine spezielle Hardwarearchitektur, die wir **FantastIC4** benannt haben. Diese macht sich komprimierte Repräsentationen der DNNs zu Nutze, um deren hocheffiziente Ausführung zu erreichen.

Unsere experimentellen Ergebnisse zeigen, dass unsere Komprimierungstechniken im Durchschnitt eine Größenreduktion von bis zu $\times 51$ erreichen und, dass wir die Anzahl der erforderlichen Operationen für die Inferenz um $\times 5.53$ reduzieren können. Außerdem konnten wir mit FantastIC4 die Flächeneffizienz (GOPS/mm²) für die Durchführung von DNNs um bis zu $\times 115$ erhöhen, zwei Größenordnungen besser als bei bisherigen Ansätzen.
Acknowledgements

First and foremost, I want thank Prof. Dr. Klaus-Robert Müller for his invaluable guidance and support throughout my Ph.D. program. Thanks to his perspectives and numerous advices on how to systematically study complex problems, I was able to learn how to perform scientific inquiry on a whole new extent of expertise. In particular, I learned to materialise the entire process into a concise article in manner that is considered of “high-quality” by the scientific community.

I also want to deeply thank Dr. Wojciech Samek, head of the Machine Learning Group at Fraunhofer Heinrich-Hertz Institute (HHI). Thanks to him, I was blessed to be part of the HHI community for nearly 5 years. Within my time at HHI, I learned how to work in a team and what it takes to produce excellent results. Moreover, I was able to travel across the world, meet renowned scientists and share interesting and exciting ideas with my peers. All of these experiences helped me to grow not only on a professional, but also a personal level. This wouldn’t have happened without Wojciechs exceptional leadership skills and trust in his colleagues and me.

I also want to express my gratitude to the department and institute leads, Prof. Dr. Thomas Wiegand, Dr. Detlev Marpe and Dr. Thomas Schierl; for giving me the opportunity to work at the HHI. Moreover, for their trust in allowing me to be part of and lead part of the DeepCABAC as well as AHEAD projects. Thanks to that I learned invaluable team-work and leadership lessons, which will help me in my on-going career.

Furthermore, I would also like to thank all my friends and colleagues from the HHI: Talmaj Marinc, Daniel Becking, Arturo Marban, Vignesh Srinivasan, Felix Sattler, David Neumann, Paul Haase, Dr. Heiner Kirchoffer, Dr. Karsten Müller and Dr. Sebastian Lapuschkin. I was able to spend an incredibly precious and unforgettable time in my almost 5 years of work together with them.

Last but not least, my gratitude also goes to my colleagues from the TU Berlin, Prof. Friedel Gerfers and Suhas Shivaprakash, for the time and effort we spend together in working on the collaborative project FantastIC4. Thanks to them I was able to gain significant know-how and knowledge about hardware systems.

Special thanks go to my beloved friends and family members that supported me throughout the years, in all the ups and downs, and are still fully supporting me in my new life adventures.
Contents

Title page i
Declaration of Authorship iii
Abstract ix
Acknowledgements xiii

1 Introduction 1
  1.1 Motivation ........................................ 1
     1.1.1 The efficiency problem of DNNs .............. 2
     1.1.2 Compact and efficient representations of DNNs to the rescue . 3
  1.2 Contributions .................................... 4
  1.3 Outline ......................................... 6
  1.4 List of publications ................................ 7

2 Background 9
  2.1 Basic concepts of Deep Learning ................. 9
     2.1.1 Definition .................................. 9
     2.1.2 Learning and Backprop ...................... 9
     2.1.3 Optimizers .................................. 10
     2.1.4 Layer types and DNN architectures .......... 10
     2.1.5 Regularization techniques .................. 10
  2.2 Communication and computation complexity of DNNs 11
     2.2.1 Anatomy of a DNN model .................... 11
     2.2.2 Inference-complexity of DNNs .............. 12
     2.2.3 Training-complexity of DNNs ............... 14
  2.3 Source coding ................................... 14
     2.3.1 Lossless coding ................................ 16
     (scalar) Huffman coding .......................... 16
     Arithmetic coding .................................. 17
     2.3.2 Universal coding ............................ 18
     CABAC ............................................ 19
     2.3.3 Lossy coding ................................ 20
     Scalar Lloyd algorithm ......................... 20
     CABAC-based RD-quantization ................. 21
  2.4 Relation between source coding and model compression 21
     2.4.1 The Minimum Description Length principle .... 21
     2.4.2 DNN model compression ..................... 22
     2.4.3 Relaxation of the DNN model compression problem into a “classical” source coding problem ............... 22
  2.5 Methods & techniques that reduce the complexity of DNNs 23
     2.5.1 Optimizations involving the DNNs topology .......... 24
3 Reducing the information content on DNNs weights

3.1 Entropy-constrained regularization of DNN training

3.2 Variational formulation of the entropy-constrained minimization objective

3.3 Entropy-constrained quantization-aware training: A generalization of the straight-through estimator

3.4 Related work

3.5 Summary

3.6 Limitations

4 Compact representations of DNNs

4.1 Introduction

4.2 DeepCABAC: A universal compression algorithm for DNNs

4.3 Experiments

4.4 Follow-up work

4.5 Summary

4.6 Limitations
5 Compact and computationally efficient representations of DNNs 65
5.1 Introduction .................................................. 65
5.2 Data structures for matrices with low entropy statistics ............... 66
  5.2.1 Previous compact matrix formats ................................ 66
    Dense formats ............................................. 66
    Sparse formats ........................................... 67
  5.2.2 Matrix formats that exploit low first-order entropy statistics ... 68
    Compressed Entropy Row (CER) format ....................... 68
    Compressed Shared Elements Row (CSER) format ............. 70
5.3 Analysis of the storage and energy complexity of data structures ... 71
  5.3.1 Definition of the computational complexity of the dot product ... 71
  5.3.2 Complexity analysis of the dense and CSR formats ............ 72
  5.3.3 Efficiency analysis of the CER and CSER formats ............ 73
  5.3.4 Connection between CSR, CER and CSER ..................... 74
5.4 Experiments ................................................... 76
  5.4.1 Experiments on simulated matrices .......................... 77
    Efficiency on different regions of the entropy-sparsity plane ... 77
    Efficiency as a function of the column size .................. 77
  5.4.2 Compressed Neural Networks without Retraining ............... 78
    Storage requirements ...................................... 79
    Number of operations ..................................... 80
    Time cost ................................................. 82
    Energy cost .............................................. 82
  5.4.3 Compressed Neural Networks with Retraining ................... 83
5.5 Summary ..................................................... 87
5.6 Limitations .................................................. 87

6 Software-Hardware co-design: Specialized hardware accelerator for com-
  pact deep neural networks 89
6.1 Introduction .................................................. 89
6.2 Related work .................................................. 90
6.3 Rationale behind FantastIC4’s design ................................ 91
  6.3.1 4bit quantization of the parameter set ....................... 92
    (Contribution 1) Increasing the computational efficiency ........ 92
    (Contribution 2) Increasing the capacity of the model .......... 93
  6.3.2 Low-entropy statistics of the parameter set .................. 93
    (Contribution 3) Saving arithmetic operations and dynamic en-
    ergy consumption ......................................... 94
    (Contribution 4) Multiple lossless compression ............... 94
6.4 FantastIC4: Efficient hardware accelerator for 4bit MLPs ............ 94
  6.4.1 Memory Controller and Input/Output Buffers .................. 95
  6.4.2 FantastIC4 Control unit .................................. 96
  6.4.3 FantastIC4 Architecture .................................. 96
6.5 Training 4bit-compact DNNs ................................... 99
6.6 Experiments ................................................... 100
  6.6.1 Experimental setup ........................................ 100
  6.6.2 Datasets & Models ....................................... 100
  6.6.3 Hardware simulation setup ................................ 100
  6.6.2 Benchmarking training of 4bit-compact DNNs ................. 100
  6.6.3 Benchmarking hardware efficiency .......................... 101
  6.6.3 Latency and throughput calculation ......................... 101
Benchmarking MLP models ........................................... 103
Comparison to previous work ...................................... 105
6.6.4 Ablation studies ................................................. 107
6.7 Summary .............................................................. 108
6.8 Limitations ........................................................... 108
7 Conclusion ............................................................... 109
A Chapter 3 - Supplemental material .............................. 113
A.1 Proof of theorem 1 ................................................... 113
A.1.1 Proof .............................................................. 114
A.1.2 Experimental validation of theorem 1 .................... 115
A.2 Details on the experimental setup of section 3.3.4 ....... 116
A.2.1 Datasets ........................................................... 116
A.2.2 Regularization techniques .................................... 117
A.2.3 Training Procedure and Optimization .................... 117
B Chapter 4 - Supplemental material .............................. 119
B.1 Experiment details ................................................. 119
B.1.1 Uniform quantization .......................................... 119
B.1.2 Lloyds hyperparameter selection .......................... 119
B.1.3 DeepCABAC’s hyperparameter selection ............... 120
DC-v1 ................................................................. 120
DC-v2 ................................................................. 120
B.2 Approximations to the Fisher Information Matrix ....... 120
B.2.1 Connection between pruning and quantization ....... 121
B.2.2 Connection between variances, Hessian, and FIM-diagonals 122
B.2.3 Hessian-weighted vs. variance-weighted quantization 122
B.2.4 More details on the experiments performed in chapter 4 123
B.3 Pseudocodes .......................................................... 123
C Chapter 5 - Supplemental material .............................. 127
C.1 Details on neural network experiments ....................... 127
C.1.1 Matrix preprocessing and convolutional layers ....... 127
Matrix decomposition ................................................. 127
Convolution layers ..................................................... 127
C.1.2 Dot product pseudocodes .................................... 127
C.2 Proof of theorems ................................................... 130
Theorem 2 .............................................................. 130
Theorem 3 .............................................................. 131
D Chapter 6 - Supplemental material .............................. 133
D.1 Experiments - A more detailed description and further results 133
D.1.1 Hand Gesture Recognition .................................. 134
D.1.2 Keyword Spotting (KWS) ................................... 134
D.1.3 Image Classification ............................................ 135
Small-scale Image Classification .................................. 135
Large-scale Image Classification ................................ 135
D.1.4 Further and more extensive results ....................... 135
D.2 Ablation studies ..................................................... 135
D.2.1 Increased capacity of FantastiC4 models ................ 135
Sensitivity of activation values vs DNN parameters ......... 135
Benefit from learning cluster centers vs fixed cluster centers . . 136

Bibliography 139
## List of Figures

2.1 Graph representation (GR) of a simple feed-forward neural network ........................................... 11
2.2 Sketch of the communication interfaces between different components of a typical inference engine ........................................................................................................ 12
2.3 The general structure of codes ........................................................................................................ 15
2.4 Arithmetic coding example .............................................................................................................. 17
2.5 Context-based Adaptive Binary Arithmetic Coding (CABAC) ............................................................ 19
2.6 Relation between source coding and model compression .................................................................. 24
2.7 Different components that affect the final resource consumption of DNNs ........................................... 25
2.8 Structured vs unstructured sparsity .................................................................................................. 27
3.1 Sketch of a two-part code for compressing data in a supervised setting based on DNNs .................. 33
3.2 MAP quantization scheme after variational entropy-constrained minimization ended .................. 37
3.3 Probability mass distribution (PMD) of the weight element values of the last layer of the LeNet-5 model after minimizing its entropy .................................................... 39
3.4 Trend of the classification loss of the VGG architecture when trained on the CIFAR10 data set under the variational entropy-constrained loss function ............................................. 40
3.5 Sketch of the entropy-constrained training method based on the STE ............................................. 44
3.6 Pareto-optimal fronts of our method EC2T vs TTQ algorithms ........................................................ 45
4.1 Sketch of the DeepCABACs compression procedure .......................................................................... 51
4.2 Distribution of the weight matrix of the last layer of VGG16 (trained on ImageNet) after uniform quantization over the range of values .............................................................. 53
4.3 DeepCABACs binarization procedure of neural networks ................................................................ 54
5.1 Sketch of the CER & CSER efficiency regions in the entropy-sparsity-plane ......................................... 75
5.2 Plots validating the CER & CSER efficiency regions from experimental simulations ........................ 78
5.3 Efficiency ratios of the CER & CSER formats compared to the dense data structure .......................... 79
5.4 Storage requirements of a compressed DenseNet after converting its weight matrices into the CER & CSER matrix formats ...................................................................................... 80
5.5 Number of operations required to perform all dot products of DenseNet, after all its weight tensors are converted into the CER & CSER matrix formats ..................................................... 81
5.6 Time spend in performing all dot products of DenseNet, after all its weight tensors are converted into the CER & CSER matrix formats ................................................................. 82
5.7 Energy costs for performing all dot products of DenseNet, after all its weight tensors are converted into the CER & CSER matrix formats ................................................................. 83
5.8 Empirical distributions of the weight matrices of different neural network architectures after compression, displayed on the entropy-sparsity plane .................................................. 84
5.9 Efficiency gains of AlexNet after converting it’s weight matrices into the CER & CSER formats .......................................................... 85

6.1 Sketch showing the difference between the MAC and ACM computational paradigms ................................................................. 92
6.2 FantastIC4 hardware system ................................................................. 95
6.3 FantastIC4 hardware architecture .......................................................... 95
6.4 CSR to bit-mask Conversion Logic ......................................................... 97
6.5 Adder-tree schematic ........................................................................... 98
6.6 Floating point multiplier logic ................................................................. 99
6.7 Comparison of different entropy-constrained training methods .......... 101
6.8 Area and power breakdown of FantastIC4s ASIC version ................. 104
6.9 Power consumption as a function of the entropy distribution ............ 107

A.1 Trend of the average log-likelihood and entropy of the continuous model and the quantized model during training of the LeNet-5 architecture ........................................................................ 115
A.2 Trend of the variance of the continuous probability distribution when the variational entropy-constrained loss function is applied .............. 116
B.1 Rate-accuracy curves for a pretrained LeNet5 model ......................... 123
D.1 Difference in sensitivity between the activations and weight parameters of the EfficientNet-B0 model .................................................. 136
D.2 Trained quantization of learned \( w_i \) vs int4-aligned \( w_i \) (powers of 2) . 137
List of Tables

3.1 Compression gains attained from training the networks under the variational entropy-constrained minimization objective. ........................................ 41
3.2 Summary of compression results from our method EC2T .................. 46
4.1 Summary of compression results from DeepCABAC ....................... 57
4.2 Comparison of DeepCABACs compression results to previous work . 59
4.3 DeepCABACs ablation study: studying the effectiveness of different quantization strategies ................................................................. 60
4.4 DeepCABACs ablation study: studying the effectiveness of the entropy coder ................................................................. 62
5.1 Energy values (in pJ) of different elementary operations for a 45nm CMOS process from (Horowitz, 2014) ......................................................... 77
5.2 Compression gains attained from applying CER & CSER to different pretrained models ................................................................. 80
5.3 Computational efficiency gains attained from applying CER & CSER to different pretrained models ................................................................. 81
5.4 Statistics of different neural network weight matrices taken over the entire network ................................................................. 83
5.5 Compression gains attained from applying CER & CSER to different sparsified models ................................................................. 86
5.6 Computational efficiency gains attained from applying CER & CSER to different sparsified models ................................................................. 86
6.1 Control States of the FantastIC4 Control Unit ............................... 94
6.2 Comparison of the FantastIC4-quantization approach vs state-of-the-art 4-bit quantization techniques ................................................................. 102
6.3 Layout results of FantastIC4’s 22nm ASIC version, running the MLP-GSC & MLP-HGR models ................................................................. 104
6.4 Performance comparison with other compression-based ASIC accelerators. Throughputs (GOPS) have been estimated w.r.t. MLP-GSC model. ................................................................. 106
6.5 Performance comparison with other ASIC accelerators specially designed for the GSC task. Ours are based on the performance of running inference of our custom MLP-GSC model on FantastIC4. ................................................................. 107
B.1 The number of clusters per layer employed for the experiment described in the experimental section V-A. ................................................................. 120
B.2 The number of clusters for the whole network as well as the $\lambda$ values used for the experiment in section V-A. ................................................................. 121
D.1 Comparison of the FantastIC4-quantization approach vs state-of-the-art 4-bit quantization techniques ................................................................. 138
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Accumulate-Multiply</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AR</td>
<td>Augmented Reality</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>Backprop</td>
<td>Backpropagation</td>
</tr>
<tr>
<td>CABAC</td>
<td>Context-based Adaptive Binary Arithmetic Coding</td>
</tr>
<tr>
<td>CER</td>
<td>Compressed Entropy Row</td>
</tr>
<tr>
<td>Conv</td>
<td>Convolutional layer</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>CSER</td>
<td>Compressed Shared Elements Row</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>CV</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>DL</td>
<td>Deep Learning</td>
</tr>
<tr>
<td>DNN</td>
<td>Deep Neural Network</td>
</tr>
<tr>
<td>ECL</td>
<td>Entropy-Constrained Lloyd</td>
</tr>
<tr>
<td>EPMD</td>
<td>Empirical Probability Mass Distribution</td>
</tr>
<tr>
<td>FC</td>
<td>Fully-Connected layer</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FIM</td>
<td>Fisher Information Matrix</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GOPS</td>
<td>Giga Operations per Second</td>
</tr>
<tr>
<td>GR</td>
<td>Graph Representation</td>
</tr>
<tr>
<td>GSC</td>
<td>Google Speech Commands</td>
</tr>
<tr>
<td>HR</td>
<td>Hand-gesture Recognition</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standard Organization</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply-Accumulate</td>
</tr>
<tr>
<td>MAP</td>
<td>Maximum-a-Posteriori</td>
</tr>
<tr>
<td>MDL</td>
<td>Minimum Description Length</td>
</tr>
<tr>
<td>MIG</td>
<td>Memory Interface Generator</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>MLP</td>
<td>Multy-Layer Perceptron</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Expert Group</td>
</tr>
<tr>
<td>NAS</td>
<td>Neural Architecture Search</td>
</tr>
<tr>
<td>NN</td>
<td>Neural Network</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PMD</td>
<td>Probability Mass Distribution</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Network</td>
</tr>
<tr>
<td>SGD</td>
<td>Stochastic Gradient Descent</td>
</tr>
<tr>
<td>SOTA</td>
<td>State-of-the-art</td>
</tr>
<tr>
<td>STE</td>
<td>Straight-Through Estimator</td>
</tr>
</tbody>
</table>
TOPS    Terra Operations per Second
List of Symbols

\( \lambda \)  
Lagrange multiplier

\( \mu \)  
Empirical probability mass distribution

\( \sigma \)  
Standard deviation

\( \nu \)  
Mean

\( \omega \)  
real value, element of a finite set

\( \mu(\cdot) \)  
Multiplication operator

\( \sigma(\cdot) \)  
summation operator

\( \gamma(\cdot) \)  
read/write operator

\( b \)  
Binary representation

\( C \)  
Code

\( D \)  
Data set

\( D(\cdot, \cdot) \)  
Distance operator

\( D_{KL}(\cdot, \cdot) \)  
Kullback-Leibler divergence

\( E[\cdot] \)  
Mean operator

\( F \)  
Fisher-Information Matrix

\( G(\cdot, \cdot) \)  
Gaussian kernel

\( H \)  
Entropy

\( L(\cdot) \)  
Code-length operator

\( \mathcal{L} \)  
Loss function

\( \mathbb{N} \)  
Natural values

\( P \)  
Probability

\( q \)  
Quantization point

\( Q(\cdot) \)  
Quantizer operator

\( Q^{-1}(\cdot) \)  
Dequantizer operator

\( Q \)  
Set of quantization points

\( \Omega \)  
Finite set of real values

\( \mathbb{R} \)  
Real values

\( W \)  
Parameter set of a particular NN

\( \mathbb{W} \)  
Set of parameters

\( \mathcal{X} \)  
Input set

\( \mathcal{Y} \)  
Label set
Dedicated to my parents, Maria Claudia Henao and Alexander Wiedemann (a.k.a. Mamita and Fahmy), for their unconditional love and support.
Chapter 1

Introduction

1.1 Motivation

Autonomous vehicles, recommendation systems, chatbots answering our inquiries, cameras recognizing our faces at superhuman level, new drugs being discovered thanks to the analysis of Big Data in the medical field, machines translating effortlessly between dozens of different languages, software generating person’s movements and voices that are indistinguishable from reality, virtual assistants on every mobile phones and smart home appliances, etc. These are all examples of disruptive artificial intelligence-based (AI) technologies that have been developed within the last decade, already impacting large sectors of our society. At the core of the fast advancements in AI lies Deep Learning (DL), a subfield of Machine Learning (ML) studying the particular class of algorithms called deep neural networks (DNNs). DNNs have enabled remarkable breakthroughs on a wide range of ML tasks such as image classification, speech recognition, object detection, natural language understanding, etc. (LeCun, Bengio, and Hinton, 2015), even making international headlines by beating human grandmasters in the games of Go (Silver et al., 2016) and Curling (Won, Müller, and Lee, 2020), challenges that experts in the field thought we were decades behind. Although the study of DNNs has its origin long back to the 60s, their recent renaissance can be attributed to three phenomena that are unique to our times:

1. Access to large amounts of data.
2. Researchers having designed novel optimization algorithms and model architectures that allow to train very deep neural networks.
3. The increasing availability of high compute resources.

All three points are key for the success of DNNs. However, they also hint to one of their main drawbacks, namely, their large resource requirements. DNN models are becoming more and more resource hungry over time, so much so that they are rapidly becoming economically, technically, and environmentally unsustainable. As stated by (Thompson et al., 2020),

"DL’s recent history has been one of achievement: from triumphing over humans in the game of Go to world-leading performance in image recognition, voice recognition, translation, and other tasks. But this progress has come with a voracious appetite for computing power... Extrapolating forward this reliance reveals that progress along current lines is rapidly becoming economically, technically, and environmentally unsustainable. Thus, continued progress in these applications will require dramatically more computationally-efficient methods, which will either have to come from changes to deep learning or from moving to other machine learning methods."
1.1.1 The efficiency problem of DNNs

It has been theoretically as well as empirically established that DNNs benefit from being overparametrized (Xu et al., 2018; Thompson et al., 2020). That is, DNN models learn better representations of the data if they are equipped with orders of magnitude more number of parameters than the number of data points they are trained on. This is in part due to applying proper training and regularization techniques (e.g. stochastic gradient descent), and due to incorporating suitable inductive bias and priors into the models network architecture (e.g. incorporating convolutional layers in computer vision tasks), which guide the optimization path to minima with excellent generalization capabilities within the data manifold. However, DNN models are notoriously inefficient learners as compared to, e.g., human beings, in the sense that they require large amounts of data corpora in order to learn meaningful representations of them. Moreover, the prediction performance of DNN models correlates well with the number of data points and, consequently, with the number of parameters of the model (Xu et al., 2018). This has caused the tendency of ML practitioners equipping the models with an ever growing number of parameters, triggering an exponential growth of their size over the past years. For instance, as of today, we have reached the point that it is becoming common practice to train models with hundreds of million of parameters on millions of data points. As stated by (Thompson et al., 2020), an example of large-scale overparameterization is the current state-of-the-art image recognition system, NoisyStudent, which has 480M parameters for imagenet’s 1.2M data points. However, the large size of DL models poses a series of challenges that difficult their deployment to production-ready environments.

Memory: The size of state-of-the-art (SOTA) DNNs typically lie in the hundreds of MBs, but can also be up to GBs as in the case of models trained for natural language processing (NLP) tasks (Xu et al., 2018). Thus, storing many models may require high memory resources, a use-case that is becoming of increasing importance since, e.g., model versioning is becoming a requirement in industrial environments. Moreover, the transmission cost of such models through communication channels with limited capacity can be significant, specially if the parameters are communicated with high frequency as in, e.g., distributed training scenarios (FL) (Sattler et al., 2020; Sattler, Müller, and Samek, 2020; Sattler, Wiegand, and Samek, 2020).

Speed: Large models are also slow in terms of execution time (high latency and small throughput) due to the high amount of computations and data movement required for performing inference. For instance, in order to translate a sentence with only 30 words, a state-of-the-art NLP model needs to execute 13 GFLOPs and takes 20 seconds on a Raspberry Pi (Wang et al., 2020). Such long latencies are prohibitive in order to provide a positive user-experience, and become much more strict in latency critical use-cases as in the autonomous driving.

Energy: As (Horowitz, 2014) showed, data movement (reading and writing from memory components) in a hardware platform incurs the highest cost in terms of energy consumption. On-chip data movement is usually an order of magnitude more expensive than performing computations, and off- to on-chip data movement (e.g. reading data from a DRAM into the chip) is up to 3 orders of magnitude more expensive. Thus, due to their large sizes SOTA DL models parameters require most often off-chip memory components, which become the major cause of energy drain. The amounts of power consumed can easily become prohibitive when deployment
1.1. Motivation

to resource constrained devices is desired, such as in mobile phones, wearables or augmented reality (AR) glasses; thus inhibiting a large number of possible applications solving critical use-cases to be created.

Cost: Large models do also incur high economical costs since they demand larger and more powerful hardware devices in order to be executed on. Moreover, this also incurs higher manufacturing cost as well as the cost of providing the necessary power supply (e.g. bigger batteries with bigger energy capacities).

Privacy: Since large amount of data and compute power is required, most often the only viable option for deploying DL-based applications is to offer cloud-based solutions to the consumers. However, this requires third-parties to have access to possibly user-sensitive data, which may be prohibitive in many use-cases.

The mentioned drawbacks greatly limit the application capabilities of DNN models for real-world use-cases. For instance, there is an ever-growing demand for deploying DNN models to edge & IoT devices such as cars, mobile phones or wearables (Howard et al., 2017; Ota et al., 2017; Wang et al., 2020), since ML models running directly on edge devices comes with a series of advantages such as an increase in positive user-experience (e.g., due to lower latencies) and higher privacy considerations (since the consumer data may not have to be send to the cloud if, on-device processing of DL models is possible). However, as mentioned above, the high resource requirements of SOTA DNN models often times prohibit their deployment to edge devices. Moreover, (Xu et al., 2018) showed that the memory-energy efficiency trend of most common hardware platforms are not able to keep up with the exponential growth-trend of the DNNs size, thus expecting them to be more and more resource hungry over time.

1.1.2 Compact and efficient representations of DNNs to the rescue

Solving the resource-efficiency problem may significantly improve AI-technologies based on DNNs, as well as open up entirely new opportunities and use-cases where DL models can be applied to. Therefore, it is paramount to study methods that can reduce the memory and computational complexity of the models.

Compression: As stated by (Wiedemann et al., 2020a), model compression is one possible paradigm to solve this problem. Namely, by attempting to maximally compress the information contained in the networks parameters we automatically leave only the bits that are necessary for solving the task. In addition, model compression can have direct practical advantages such as reduced communication and compute cost (Sze et al., 2020; Deng et al., 2020; Sattler, Wiegand, and Samek, 2020). In fact, the Moving Picture Expert Group (MPEG) of the International Organization of Standards (ISO) has recently issued a call on neural network compression (MPEG Requirements, 2019), which stresses the relevance of the problem and the broad interest by the industry to find practical solutions.

Efficient computation: A related field of study is the design and analysis of methods that try to reduce the computational complexity involved in executing DNN models (Sze et al., 2020; Deng et al., 2020; Wiedemann, Müller, and Samek, 2020). For instance, this may be methods that try to replace or minimize the number of expensive operations such as multiplications, or optimize the computational flow so
that redundant computations are not performed (e.g., skipping multiplications and summations with 0 values).

**Specialized hardware:** In parallel, are the works that focus on designing specialized hardware architectures for DNN models. They are tailored to the special computations required by DNN models and exploit the particular properties in them in order to increase their execution efficiency (Sze et al., 2020).

In particular, the execution efficiency of DNNs can be optimized from all three fronts simultaneously, by e.g. following a software-hardware co-design paradigm. Such approaches can attain highest gains in efficiency, in many cases orders of magnitude more than their counterparts, consequently permitting the deployment of SOTA DNN models to devices with very tight resource constraints such as microcontrollers.

### 1.2 Contributions

The main research focus in this thesis is on developing methods for compressing DNN models and designing algorithms that can efficiently process compact representations of them at inference time. Thus, methods that regard the efficient training of DNN models are out of the scope of this thesis.

The particular contributions presented in this thesis can be summarized as follows:

- In chapter 3 we present two novel regularization techniques that allow to explicitly minimize the information entailed in DNNs parameters during training, where the information is characterized as the empirical first-order entropy of the parameter elements. One method consists on deriving a variational formulation of the entropy-constrained objective function, that is differentiable and can be minimized by applying scalable Bayesian techniques. The other method is based on the straight-through estimator, which calculates surrogate gradients that guide the optimization path of the models into minima with low information content. We describe the advantages and disadvantages of both methods and show that state-of-the-art DNNs can be trained to be highly compact with these techniques.

  - **Mr. Wiedemann’s contribution:** He researched, designed and developed the presented methods. He also provided the theoretical analysis of the methods (e.g., theorem 1), and devised the experiments and ablation studies presented in the chapter.

  - **Co-author’s contributions:** Dr. Arturo Marban implemented the experimental setup of section 3.2.4 using PyTorch and provided helpful and critical feedback regarding the results and analysis of the experiments. He also provided valuable feedback in the experimental analysis of section 3.3.4. Daniel Becking implemented the experimental setup explained in section 3.3.4 using PyTorch and provided helpful feedback regarding the results and analysis of the experiments. All authors, Dr. Arturo Marban, Daniel Becking, Dr. Wojciech Samek and Prof. Klaus-Robert Müller; provided valuable feedback and helped shape the research, analysis and content of the manuscripts (Wiedemann et al., 2019; Marban et al., 2020) on which the chapter is based on.
1.2. Contributions

- In chapter 4 we present DeepCABAC, a state-of-the-art compression technique that produces maximally compact representations of DNNs. It applies CABAC to the DNNs weights, a powerful lossless compression algorithm widely applied in media compression and entailed in the H.264 & H.265 video compression standard. In the context of this thesis we focus on the thorough study of the advantages and disadvantages of applying CABAC for neural network compression, and consequently in the development of different preprocessing and quantization techniques that boost the compression performance as well as the efficiency of the algorithm.

  - Mr. Wiedemann’s contribution: He designed the rate-distortion quantization schemes presented in the chapter and respective manuscript (Wiedemann et al., 2020a), which are a fundamental component of DeepCABAC’s compression engine. Moreover, he proposed the training schemes that render the DNNs to be amenable for compression with DeepCABAC’s engine, which lead to the state-of-the-art compression results presented in the chapter. He also designed the experiments presented in the ablation studies, helping the analysis and understanding of the effectiveness of each component of DeepCABAC’s engine. Finally, he wrote the manuscript this chapter is based on almost in its entirety (Dr. Heiner Kirchhoffer contributed in parts of the introduction and section II of the text in the article).

  - Co-author’s contributions: Dr. Heiner Kirchhoffer, Paul Haase & Stefan Matlage, developed the lossless compression engine rooted in DeepCABAC. They also implemented an efficient and robust version of the algorithm in C++, including the quantization scheme. Talmaj Marinc’s, David Neu mann’s and Dr. Arturo Marban’s contributions focused in the engineering side of the project, that is, in the implementation of fast, large-scale experiments and benchmarks that can be performed robustly across different type of DNN models and compression settings. They also implemented the training of the DNN models as well as the sparsification techniques described in the experimental section of the chapter. Dr. Tung Nguyen, Prof. Heiko Schwarz, Prof. Thomas Wiegand, Prof. Detlev Marpe and Dr. Wojciech Samek provided valuable feedback and critical analysis during the development of the project. They also provided necessary resources for the successful development and experimental execution of the project, such as access to large amounts of computing power (GPUs, cloud services, etc.). All authors provided valuable feedback and helped shape the research, analysis and content of the manuscript (Wiedemann et al., 2020a).

- In chapter 5 we design novel matrix formats suitable for representing DNNs weights with low-entropy statistics. These new matrix formats, named Compressed Entropy Row (CER) and Compressed Shared Elements Row (CSER), do not only offer higher compression gains than the most commonly used alternatives, but also reduce further the computational complexity for performing the dot product.

  - Mr. Wiedemann’s contribution: He researched and developed the presented methods and algorithms in their entirety.
Chapter 1. Introduction

- **Co-author’s contributions**: Dr. Wojciech Samek and Prof. Klaus-Robert Müller, provided critical feedback and helped shape the research, analysis and content of the manuscript.

- Finally, in chapter 6 we propose a software-hardware co-design paradigm for training and running highly efficient and compact DNNs. On the one hand, we develop a novel training method that makes DNNs robust to 4-bit quantization while encouraging low-entropy statistics of the weight parameters simultaneously. On the other hand, accompanied by Daniel Becking from Fraunhofer HHI & Suhas Shivapakash from TU Berlin, we design a novel hardware architecture called FantastIC4 that is able to leverage the 4-bit + low-entropy statistics of the weight parameters in order to minimize the time, area, memory and energy requirements for running the models.

- **Mr. Wiedemann’s contribution**: He designed the information processing scheme of the hardware architecture in its entirety. That is, he designed the decoders functionality, how the ACM computational paradigm ought to be implemented on a hardware level (e.g., with adder-trees and stationary activation values) and how the on-chip data movement should be. In addition, he designed the novel entropy-constrained training algorithm that renders the DNN models amenable for efficient execution in FantastIC4’s hardware. Moreover, he designed and devised all the experiments and ablation studies presented in the chapter. Finally, he is the main contributor to the text of the article (Wiedemann et al., 2021), on which this chapter is based on.

- **Co-author’s contributions**: Suhas Shivapakash’s main contribution lied in the implementation of the hardware architecture and the choice of its hardware components, so that maximal efficiency can be achieved. He also contributed to major parts of the text of the article, such as section 5 in the article (section 6.4 in the thesis). Daniel Becking’s main contribution focused on the implementation of the experimental setup and entropy-constrained training of the considered DNN models. All authors, Suhas Shivapakash, Daniel Becking, Pablo Wiedemann, Dr. Wojciech Samek, Prof. Friedel Gerfers and Prof. Thomas Wiegand; provided critical feedback and helped shape the research, analysis and content of the manuscript.

1.3 Outline

The outline of the thesis is as follows:

- **Chapter 2**: In chapter 2 we start by introducing DNNs, their architectures, training methods and common datasets used for benchmarking different ML tasks. Moreover, we give an extensive overview of the relevant literature on the topic of efficient processing of DNNs, and stress our contributions to this topic. In addition, we also introduce the field of source coding, and describe a mathematical framework that unites both worlds under one common paradigm.

- **Chapter 3**: Motivated by the Minimum Description Length (MDL) principle, we derive an entropy-constrained regularization function which explicitly constraints the information content of the weight parameters of the DNN during training. This results in networks that can be highly compressed after training.
Most of the content of this chapter are based on two conference publications, where the author of this thesis are first authors of the publication (Wiedemann et al., 2019), or one of the main contributors (Marban et al., 2020).

- **Chapter 4:** We apply state-of-the-art video coding techniques to maximally compress the size of DNNs parameters. Concretely, our coding technique named DeepCABAC is based on applying Context-based Adaptive Arithmetic Coding (CABAC), which is a powerful lossless compression engine widely used in the video coding standards such as H.264 & H.265. The content of this chapter is heavily based on the journal publication (Wiedemann et al., 2020a), where the author of this thesis are the first author of the publication.

- **Chapter 5:** We derive novel tensor formats that are specially suitable for weight parameters that exhibit low-entropy statistics. The representations called Compressed Entropy Row (CER) & Compressed Shared Elements Row (CSER) do not only compress more the weight parameters than previous techniques, but do also reduce the computational complexity involved in executing the dot product operation. The content of this chapter is heavily based on the journal publication (Wiedemann, Müller, and Samek, 2020), where the author of this thesis are the first author of the publication.

- **Chapter 6:** We present FantastIC4, a hardware accelerator that is able to leverage the low-entropy statistics of the weight parameters in order to significantly increase the inference efficiency of compact models. The content of this chapter is heavily based on a journal publication which is still in review process, where the author of this thesis are one of the first authors of the work presented.

- **Chapter 7:** In our last chapter we summarize our findings of the thesis and discuss further future work on this topic.

Overall, the topics discussed in this thesis focus around the information theoretical measure of entropy (Shannon, 1948), how it can be characterized and optimized in DNNs models, and furthermore exploited to reduce their communication as well as compute cost.

### 1.4 List of publications

In the following we present a list of contributions made by the author to the field of efficient processing of DNNs.

The focus of this thesis lie on the following publications:

- Simon Wiedemann et al. (2019). “Entropy-Constrained Training of Deep Neural Networks”. In: *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. DOI: 10.1109/IJCNN.2019.8852119


Chapter 1. Introduction

- Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek (2020). “Compact and Computationally Efficient Representation of Deep Neural Networks”. In: IEEE Transactions on Neural Networks and Learning Systems 31.3, pp. 772–785. DOI: 10.1109/TNNLS.2019.2910073


Other publications and contributions include:

- Felix Sattler et al. (2019). “Sparse Binary Compression: Towards Distributed Deep Learning with minimal Communication”. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. DOI: 10.1109/IJCNN.2019.8852172

- Felix Sattler et al. (2020). “Robust and Communication-Efficient Federated Learning From Non-i.i.d. Data”. In: IEEE Transactions on Neural Networks and Learning Systems 31.9, pp. 3400–3413. DOI: 10.1109/TNNLS.2019.2944481


Chapter 2

Background

In this chapter we will analyse in detail the sources of resource consumption that emerge from executing DL models. This will help us understand and categorize the different approaches and techniques that are being studied in the literature, all having the common goal of increasing the execution efficiency of the models.

Since many of the techniques described in this thesis are motivated by the source coding literature, we will also provide a quick introduction to this topic and formulate a mathematical relationship between the two fields.

We start the chapter by refreshing some of the basic concepts from the field of Deep Learning.

2.1 Basic concepts of Deep Learning

2.1.1 Definition

Deep Neural Networks (DNNs) belong to a particular class of Machine Learning (ML) algorithms that were originally inspired by the information processing system of brains (LeCun, Bengio, and Hinton, 2015; Goodfellow, Bengio, and Courville, 2016). They consist of artificial neurons organized in so called layers, with specific inter-layer connectivity patterns. The strength of the connectivity between the neurons is determined by the DNNs parameters, and the activation functions define how the neurons respond to incoming signals from the neurons it is connected to.

In mathematical terms, DNNs can be represented by acyclic graphs defining how information is transmitted between neurons/nodes. In one of their most simple forms, they can be formulated as concatenations of affine transformations followed by non-linear transformations. The parameters would then be represented by the tensors of the affine transformations, and the activation functions by the non-linear functions assigned to each output of the transformation.

2.1.2 Learning and Backprop

Learning, consists then in the process of determining a specific parameter configuration that solve a particular task as good as possible (e.g., image classification). Since DNNs are usually equipped with many millions of parameters, scalable, gradient-based optimization techniques are usually employed in order to iteratively update the parameters. In practice, arguably the most common technique employed for computing the gradients is the Backpropagation (backprop) algorithm (Rumelhart, Hinton, and Williams, 1986) since, due to its high parallelization capabilities, it is most often computationally feasible when applied to state-of-the-art, general purpose processor units such as CPUs or GPUs.
2.1.3 Optimizers

A plethora of optimization techniques have been studied and proposed in the literature over the past years (Sun et al., 2020). To recall, gradient-based optimization techniques consist of iteratively changing the parameters in the negative direction of the gradient by a small step. The step-size is also sometimes called the learning rate.

One of the most commonly applied techniques is Stochastic Gradient Decent (SGD), which consist of randomly sampling mini-batches of the training data, computing the gradients and subsequently updating the parameters accordingly at each iteration. However, we stress that other techniques are also commonly applied and worth mentioning, such as ADAM (Kingma and Ba, 2015), which tries to overcome the problem of having to manually set the learning rate by proposing an automated, adaptive update schedule based on the local information geometry of the training data.

2.1.4 Layer types and DNN architectures

A particular connectivity pattern between the neurons is also referred as the architecture of the DNN.

Arguably, one of the most trivial architectures are so-called Multi-Layered Perceptrons (MLP), consisting of a sequence of layers with the neurons of one layer connected to all neurons of its preceding layer. This type of inter-layer connectivity pattern is also referred as fully-connected (FC) layers in the literature.

However, over the past years it has been established that incorporating prior knowledge into the architecture is a highly effective method for increasing the prediction performance of DNNs. For instance, a significant boost in prediction performance can be attained across computer vision tasks if one incorporates convolution operations in the DNN layers. These type of layers are called convolutional (conv) layers. This has led to researchers and ML practitioners investigating and proposing a wide set of different type of architectures and layer-types, all with a unique connectivity pattern. To the interested reader, we refer to (Han, 2017) for a comprehensive survey of some of the most commonly used DNN architectures and layers as of today.

2.1.5 Regularization techniques

It is also common practice to apply regularization techniques in order to prevent over-fitting and increase generalization performance of the models. Here it is worth mentioning Dropout (Srivastava et al., 2014), a regularization technique which consists in stochastically turning off a percentage of neurons during training. Dropout has demonstrated to be very effective in preventing memorization in practice, while offering a very light-weight and scalable implementation simultaneously. Other methods such as $L_p$-regularization of the weights are also applied by the community.

For a more thorough overview, analysis and explanation of the here introduced concepts, we refer to the comprehensive introductory deep learning book (Goodfellow, Bengio, and Courville, 2016).
2.2 Communication and computation complexity of DNNs

In the following we will provide an in-depth analysis of the sources of communication and execution complexity that emerge from processing DNNs.

2.2.1 Anatomy of a DNN model

Firstly, we will recall the different components that constitute a particular DNN model. In the highest level of abstraction, DNNs are composed by two entities:

1. **Graph representation (GR)** which describes the topology of a DNN model. That is, GR describes the computation and the control flow of the data as it is processed by a particular neural network model, where each of its nodes indicates the processing of the input as defined by an elementary operation. Naturally, the set of elementary operations is defined a priori. Some examples of elementary operations may be additions, element-wise multiplications, tensor dot products such as matrix multiplications or convolutions, as well as operations more specific to DL such as pooling, batch-normalization, dropout, etc.

2. **Parameter set** that describe the particular configuration of the respective nodes. E.g. the set may contain the values of the weight tensors, bias and batchnorm parameters, etc.

An example of a simple GR is depicted in figure 2.1 which shows the computational flow of a feed-forward neural network specifically designed to classify hand-written digits from the MNIST dataset (LeCun and Cortes, 2010). It is composed by a series of convolutional (conv) and fully-connected (fc) layers with ReLU activation functions, connected by pooling and reshaping operations in between. The GR is restricted to only describe the connectivity of the operations, as well as the dimensions of the specific input and parameter tensors involved at each node. However, it does not contain information regarding the particular parameter values in the conv and fc layers. As described above, the latter information is usually described in a separate entity (e.g., file, directory, key, etc.), which we refer as the *parameter set* of the model.

In most real-world scenarios the processing of the information of the parameter set dominates the complexity of DNN models, since usually the description of the
Figure 2.2: High-level sketch of the communication interfaces between different components of a typical inference engine. Usually, these are von Neumann compute architectures which have a small local memory unit where (partially) the model, input and output data is stored, which communicates directly with the computation engine. The computation engine performs computations according to the instruction module dictated by the network. Since usually the local memory unit does not have the necessary capacity to store all data elements involved in the entire inference procedure, it communicates with an external memory unit with sufficient capacity to store all necessary information for performing the entire inference procedure, e.g. model, input and intermediate outputs.

GR is significantly smaller and therefore simpler in comparison. For instance, in the above example, the information entailed in the GR is about 2.5KB, which is only 9% of the total size of the model if we include the information of the parameter set\(^1\). This difference becomes more pronounced for larger models designed for solving more complex tasks, where the contribution of the GR to the total size is most often lower than 0.1%. Hence, since over 99% of the cost is spent in processing the information entailed in the parameter set, our main focus lies on reducing its complexity. Thus, optimizations of the GR are out of the scope of this thesis.

2.2.2 Inference-complexity of DNNs

Due to pedagogical reasons, in the following we will describe in a relatively simplistic, conceptually high-level manner the inner workings of a typical inference engine of DNNs. The goal is to shed light into the components that incur highest costs in terms of resource consumption when processing DNNs. We will assume a von Neumann architecture for the inference engine as depicted in figure 2.2, since it is the most common hardware architecture employed in the literature as well as in the industry.

\(^1\)The respective sizes were calculated with regards to the ONNX format of DNN models (Bai, Lu, Zhang, et al., 2019)
Whenever it is asked to perform inference on a model (or particular layer) it is implicitly requested to send the input as well as the model data (e.g. partial components of the parameter set as well as the respective GR) to the compute engine, which computes the outputs as specified by the model architecture. Notice, that the GR of the model plays the role of determining the type of computation that needs to be performed between the parameter set and the inputs/activations and of scheduling these computations in the right order. Thus, the information entailed in the GR of a particular DNN model can be interpreted as being part of the instruction set of the inference engine. This fact stresses the differences between the two entities entailed in a DNN model description, reflected in the manners their particular information content is processed and analyzed.

Since usually the input/activations and the parameter set are big in size, a memory unit with high capacity is required to store this data. However, reading data from and writing data to large memory units is highly expensive in terms of energy and time consumption. For instance, (Horowitz, 2014) showed that the energy consumption involved in read/write operations from a DRAM can be up to two orders of magnitude higher than read/write operations from a smaller sized SRAM, and up to three orders of magnitude higher than compute (addition & multiplications) operations. Therefore, a common optimization strategy is to integrate different memory units of different sizes into the engine, where smaller sized memories with higher read/write operation efficiency are positioned closer to the compute engine. Subsequently, the data movement from and to the compute engine is optimized so that the larger memory units are utilized as least as possible. Figure 2.2 sketches conceptually this interplay between different memory units. In the sketch we make explicitly a distinction between external memory and local memory, since the external memory may be located in an entirely different physical place, such as on a hard disk or even on a server with only wireless communication capabilities.

Apart from time and energy costs involved in the data movement from and to the compute engine, DNN models with large amounts of parameters are also very compute-hungry. For instance, as stated by (Han, 2017), ResNet-152 requires 22.6GOPS for performing inference. Running ResNet-152 in a self-driving car with 8 cameras at 1080p 30 frames/sec requires the hardware to deliver 22.6GOPS × 30fps × 8 × 1920 × 1280/(224 × 224) = 265 Teraop/sec computational throughput. Building and incorporating hardware component that satisfy such requirements on self-driving cars is not only technically challenging, but also economically unfeasible.

With figure 2.2 in mind, it becomes relatively trivial to recognize that compression may help boost the efficiency of inference. For instance, by compressing the parameter set, less amount of data need to be accessed and communicated from and to large memory units, or even directly from and to the compute engine. Moreover, lower amount of memory requirements are needed and therefore memory units with lower capacities can be employed instead, as such reducing the physical resource requirements for executing the models which directly translates to lower energy consumption and production costs. In addition, if we consider scenarios where model parameters are communicated through an expensive communication channel such as in a wireless setting, e.g. as in model updates or in distributed training scenarios (Sattler, Wiegand, and Samek, 2020), compression naturally minimizes the communication costs involved in such processes. Finally, compression can also help reduce the computational cost involved in processing a compact representation of the parameter set, thus reducing the total number of operations required for performing inference (more on this in chapter 4).
### 2.2.3 Training-complexity of DNNs

These efficiency-challenges transfer to an even greater extend to training of DNNs. Firstly, the GR expands in order to inform about the computation of the gradients of the DNNs parameters. In some cases, its size can grow to an extend where the resource consumption incurred from processing it can not be neglected anymore. Secondly, the gradient information adds significant amount of memory and compute requirements into the system, so much so that it roughly accounting for 2/3 of the resource-consumption of the training algorithm. This increases the challenges involved in optimizing data movement and computation of the algorithm. Thirdly, the training algorithm itself usually requires the processing of the DNNs for several thousands of iterations until convergence.

Thus, optimizing the training-efficiency of DNNs comes with its unique set of challenges, which is the reason why it is becoming its own field of study (Sze et al., 2020; Deng et al., 2020; Wiedemann et al., 2020b; Sattler, Wiegand, and Samek, 2020).

In order to narrow the scope of this dissertation, we focus on the problem of reducing the inference-complexity of trained DNN models. There are a wide set of real-world applications where inference complexity dominates the costs involved in processing DNNs. For instance, in use-cases where the models need to be trained only once but they have billions of inference-requests/day (e.g., sentiment analysis of text messages in twitter or chat rooms), or in real-time applications where predictions of DNN models need to satisfy tight latency constraints (e.g., real-time recommendations or real-time user-behaviour). Moreover, in many edge-applications the models can be trained offline on powerful compute engines, but are deployed to embedded devices with tight memory and compute constraints (e.g., mobile phones, micro-controllers, etc.). Finally, many of the analysis and methods developed for reducing inference-complexity (e.g., pruning and/or quantization) may also be applicable for reducing the training-complexity.

Although we have pointed out some of the potential benefits of applying data compression as a means to reduce communication- and inference-complexity, we have not yet stated any theoretical framework that thoroughly formulates it. This belongs to the realm of source coding theory, which we will briefly introduce in the following subsection.

### 2.3 Source coding

Source coding is a subfield of information theory that studies the properties of so called *codes*. These are mappings that assign a binary representation and a reconstruction value to a given input element. Figure 2.3 depicts their most common structure. They are comprised of two parts, an *encoder* and a *decoder*. The encoder is a mapping that assigns a binary string of finite length $b$ to an input element $w$. In contrast, the decoder assigns a reconstruction value $q$ to the corresponding binary representation. We will also sometimes refer to $q$ as a *quantization point*. Furthermore, it is assumed that the output elements $b$ and $q$ of the code $C$ are elements of finite countable sets, and that there is a one-to-one correspondence between them. Therefore, without loss of generality, we can decompose the encoder into a *quantizer* and a *binarizer*, where the former maps the input to an integer value $Q(w) = i \in \mathbb{Z}$, and the latter maps the integers to their corresponding binary representation $B(i) = b$. 

---

Disclaimer: Major portions of the content of section 2.3 are reprinted and with permission from (Wiedemann et al., 2020a), ©2020 IEEE.
2.3. Source coding

Figure 2.3: The general structure of codes. Firstly, the encoder maps an input sample \( w \) from a probability source \( P(w) \) to a binary representation \( b \) by a two-step process. It quantizes the input by mapping it to an integer \( i = Q(w) \). Then, the integer is mapped to its corresponding binary representation \( b = B(i) \) by applying a binarization process. The decoder functions analogously maps the binary representation back to its integer value by applying the inverse \( B^{-1}(b) = i \) and assigns a reconstruction value (or quantization point) \( Q^{-1}(i) = q \) to it. We stress that \( Q^{-1} \) does not have to be the inverse of \( Q \). ©2020 IEEE.

Analogously for the decoder. Naturally, it follows that the binarizer is always a bijective map, thus \( (B^{-1} \circ B)(i) = i \).

We also distinguish between two types of codes, the so called lossless codes and lossy codes. They respectively correspond to the cases where \( Q \) is either bijective or not, thus, the latter implies that information is lost in the coding process. Therefore, we stress that the map \( Q^{-1} \) does not necessarily have to be the inverse of \( Q \)!

After establishing the basic definition of codes we will now formalize the source coding problem. In simple terms, source coding studies the problem of finding the code that maximally compresses a set of input samples, while maintaining the error between the input and reconstruction values under an error tolerance constraint.

Or more precisely: let \( W \subset \mathbb{R}^n \) be a given input set and let \( P(w) \) be the probability of an element \( w \in W \) being sampled. Then, find a code \( C^* \) that

\[
C^* = \arg \min_C \mathbb{E}_{P(w)} [D(w, q) + \lambda L_C(b)]
\]

where \( b = (B \circ Q)(w) \) and \( q = (Q^{-1} \circ Q)(w) \). \( D \) is some distance measure and \( L_C \) is the length of the binary representation \( b \). We will refer to \( L_C(\cdot) \) as the code-length of a sample, and to \( D \) as the distortion between \( w \) and \( q \). \( \mathbb{E}_{P(\cdot)} \) denotes expectations as taken by the probability distribution \( P \). \( \lambda \in \mathbb{R} \) is the Lagrange multiplier that controls the trade-off between the compression strength and the error incurred by it.

Minimization objectives of the form (2.1) are called rate-distortion objectives in the source coding literature. However, solving the rate-distortion objective for a given input source is most often NP-hard, since it involves finding optimal quantizers \( Q \), binarizers \( B \) and reconstruction values \( Q^{-1} \) from the space of all possible maps.
Chapter 2. Background

However, concrete solutions can be found for special cases, in particular in the lossless case. In the following we will review some of the fundamental theorems of source coding theory and introduce state-of-the-art coding algorithms that produce binary representations with minimal redundancy.

2.3.1 Lossless coding

Lossless coding implies that the quantizer $Q$ is bijective and therefore $q = (Q^{-1} \circ Q)(w) = w \ \forall w$. Thus, $D(w, q) = 0 \ \forall w \in \mathcal{W}$ in (2.1) and the rate-distortion objective simplifies into finding a binarizer $B^*$ that maximally compresses the input samples. Hence, throughout this subsection we will equate the general code $C$ with the binarizer $B$ and refer to it accordingly.

Information theory already makes concrete statements regarding the minimum information contained in a probability source. Namely, Shannon in its influential work (Shannon, 1948) stated that the minimum information required to fully represent a sample $w$ that has probability $P(w)$ is of $-\log_2 P(w)$ bits. Consequently, the entropy $H_P(\mathcal{W}) = \sum_{w \in \mathcal{W}} -P(w) \log_2 P(w)$ states the minimum average number of bits required to represent any element $w \in \mathcal{W} \subset \mathbb{R}^n$. This implies that

$$H_P(\mathcal{W}) \leq \bar{L}_C(\mathcal{W}), \ \forall C$$

(2.2)

where $\bar{L}_C(\mathcal{W}) = \sum_{w \in \mathcal{W}} P(w)L_C(w)$ is the average code-length that any code $C$ assigns to each element $w \in \mathcal{W}$. Eq. (2.2) is also referred as the fundamental theorem of lossless coding.

Fortunately, from the source coding literature (Wiegand and Schwarz, 2011) we know of the existence of codes that are able to reach the average code-length, up to only 1 bit of redundancy to the theoretical minimum. That is,

$$\exists C : H_P(\mathcal{W}) \leq \bar{L}_C(\mathcal{W}) < H_P(\mathcal{W}) + 1$$

(2.3)

Moreover, we even know how to build them.

Before we start discussing in more detail some of these codes, we want to recall an important property of joint probability distributions. Namely, due to their sequential decomposition property, we can express the minimal information entailed in an output sample $w \in \mathbb{R}^n$ as

$$- \log_2 P(w) = - \sum_{j=0}^{n-1} \log_2 P(w_j|w_{j-1}, \ldots, w_0)$$

That is, we can always interpret a given input vector as an $n$-long random process and encode its outputs sequentially. As long as we know the respective conditional probability distributions, we can optimally encode the entire sequence. Respectively, we denote with $w_j$ the scalar value of the $j$-th dimension of $w$ (or equivalently $j$-th output of the random process). Also, we denote with $\mathcal{W}_s$ the set of possible scalar inputs, where $w_j \in \mathcal{W}_s, \forall j$.

{(scalar) Huffman coding}

One optimal code is the well known Huffman code (Huffman, 1952). However, Huffman codes can be very inefficient in practice since the Huffman-tree grows very quickly for large input dimensions $n$. Therefore, most often scalar Huffman codes are used instead. These codes consider 1-dimensional inputs only and are therefore


suboptimal, i.e., they produce redundant binary representations. Concretely, they produce average code-lengths of

$$H_P(\mathcal{W}) \leq \bar{L}_{\text{SH}}(\mathcal{W}) < H_P(\mathcal{W}) + n$$

where $\bar{L}_{\text{SH}}(\cdot)$ is the average code-length produced by the scalar Huffman code on an entire input sequence $w \in \mathcal{W} \subset \mathbb{R}^n$. Hence, if $n$ is large (e.g., in the order of hundreds of millions of values as in the case of DNN parameter sets), then the resulting binary representation of the entire sequence of values may have a high number of redundant bits. We described its pseudocode in the appendix B.

**Arithmetic coding**

A concept that approaches the joint entropy $H(\mathcal{W})$ of eq. (2.3) in a practical and efficient manner is arithmetic coding. It consist of expressing a particular sequence of samples $w_0, w_1, \ldots, w_{n-1}$ of an $n$-long random process as a so-called coding interval. An overview of the idea is given in the following.

Let $[L_j, L_j + R_j]$ be the coding interval before encoding symbol $w_j$ and let $L_0 = 0$ and $R_0 = 1$. Encoding of a symbol $w_j$ corresponds to deriving a coding interval $[L_{j+1}, L_{j+1} + R_{j+1})$ from the previous interval $[L_j, L_j + R_j)$ as follows. Subdivide $[L_j, L_j + R_j)$ into one subinterval for each element $w_j$ of $\mathcal{W}$ so that the interval width is given as

$$R_j \cdot P(w_j | w_{j-1}, w_{j-2}, \ldots, w_0)$$

for a given sequence of (already sampled) values $w_{j-1}, w_{j-2}, \ldots, w_0$, and arrange the subintervals so that they are non-overlapping and adjacent. The subinterval associated with the sample $w_j$ to be encoded becomes the new coding interval $[L_{j+1}, L_{j+1} + R_{j+1})$. Encoding of $n$ symbols yields the coding interval $[L_n, L_n + R_n)$ and the sequence of symbols $w_0, w_1, \ldots, w_{n-1}$ can be reconstructed (in the decoder) when an arbitrary value inside of this coding interval is known. Figure 2.4 exemplifies this procedure for a binary random process. Interestingly, the width of the coding interval $R_n = P(w_0, w_1, \ldots, w_{n-1})$ equals the probability of sequence $w_0, w_1, \ldots, w_{n-1}$. As the minimum achievable code length for encoding of the $n$ symbols is known to be $- \log_2(R_n)$, the location of interval $[L_n, L_n + R_n)$ needs to be signaled to the decoder in a way so that the number of written bits gets as close to $- \log_2(R_n)$ as possible.
The basic encoding principle is as follows. Derive an integer \( k \) so that
\[
2^k \leq \frac{R_n}{2} < 2^{1-k}
\] (2.4)
holds. Subdivide the unit interval \([0, 1)\) into \( 2^k \) (adjacent and non-overlapping) subintervals \([q2^k, (q+1)2^k)\) of width \( 2^k \). Equation (2.4) guarantees that one of the intervals \([q2^k, (q+1)2^k)\) is fully contained in the coding interval (regardless of the exact location \(L_n\) of interval \([L_n, L_n + R_n)\)) and if the decoder knows this interval, it can unambiguously identify \([L_n, L_n + R_n)\). Consequently, the index \(q\) identifying this interval is written to the bitstream using \( k \) bits. Equation (2.4) can be rewritten as
\[
k < - \log_2(R_n) + 2
\] (2.5)
which shows that the an ideal arithmetic coder only requires up to two bits more than the minimum possible code length for a sequence of length \( n \).

### 2.3.2 Universal coding

In the previous subsection we learned that there exist codes that are able to produce binary representations of (almost) minimal redundancy (e.g. arithmetic codes). However, we implicitly assumed that the decoder knows the joint probability distribution of the input source. This is not the case in many real world scenarios. Hence, in such cases one usually relies on so called universal codes (Grünwald and Rissanen, 2007). They basically apply the following principle:

1. start with a general, prior probability model \( P_{\text{Dec}} \),
2. update the model upon seeing data,
3. encode the input samples with regards to the updated probability model.

Thus, the theoretical minimum of universal codes are lower bounded by the decoder’s probability estimate. Concretely, let \( P_{\text{Dec}} \) be the decoder’s estimate of the inputs probability model, then the minimum average code-length that can be achieved is
\[
\bar{L}_C(W) \geq H_{P,P_{\text{Dec}}}(W) = H_P(W) + D_{KL}(P \mid\mid P_{\text{Dec}})
\]
with \( H_{P,P_{\text{Dec}}}(W) = - \sum_{w \in W} P(w) \log_2 P_{\text{Dec}}(w) \) being the cross-entropy and \( D_{KL} \) the Kullback-Leibler divergence. Hence, a lossless code can only create binary representations with minimal redundancies if and only if its decoder’s probability model matches the input source’s. In other words, the better its estimate is, the more compact it can encode the input samples.

In general, a universal lossless code should have the following properties:

- **Universality**: The code should have a mechanism that allows it to adapt its probability model to a wide range of different types of input distributions, in a sample-efficient manner.
- **Minimal redundancy**: The code should produce binary representations of minimal redundancy with regards to its probability estimate.
- **High efficiency**: The code should have high coding efficiency, meaning that encoding/decoding should have high throughput.
2.3. Source coding

CABAC

Context-based Adaptive Binary Arithmetic Coding (CABAC) is a universal lossless codec that encodes an $n$-long sequence of 1-dimensional values by: 1) representing each unique value by a binary string that corresponds to traversing a particular path on a predefined decision tree, 2) assigning to each decision (or bin) a probability model (context model) and updating these upon encoding/decoding data, and 3) applying a binary arithmetic coder in order to encode/decode each bin. ©2020 IEEE.

CABAC

Context-based Adaptive Binary Arithmetic Coding is a form of universal lossless coding that fulfils all the above properties. It offers a high degree of adaptation, optimal code-lengths, and a highly efficient implementation. It was originally designed for the video compression standard H.264/AVC (Marpe, Schwarz, and Wiegand, 2003), but it is also an integral part of its successor H.265/HEVC. It is well known to attain higher compression performance as well as higher throughput than many other entropy coding methods (Marpe and Wiegand, 2003). In short, it encodes each input sample by applying the following three stages:

1. **Binarization:** Firstly, it predefines a series of binary decisions (also called bins) under which each unique input sample element (or symbol) will be uniquely identified. Thus, it builds a predefined binary decision tree where each leaf identifies a unique input value.

2. **Context-modeling:** Then, it assigns a binary probability model to each bin (also named context model) which is updated on-the-fly by the local statistics of the data. This enables CABAC to model a high degree of different source distributions.

3. **Arithmetic coding:** Finally, it employs an arithmetic coder in order to optimally and efficiently code each bin, based on the respective context model.

Notice that, in contrast to Huffman codes, CABAC’s encoder does **not need to encode its probability estimates**, since the decoder is able to analogously update its context models upon sequentially decoding the input samples. Codes that have this property are called *backward-adaptive* codes. Moreover, it is able to take *local correlations* into account, since the context models are updated in an autoregressive manner by the local statistics of the data.
2.3.3 Lossy coding

In contrast to lossless coding, information is lost in the lossy coding process. This implies that the quantizer $Q$ is non-invertible, and therefore $\exists w : D(w, q) \neq 0$. An example of a distortion measure may be the mean-squared error $D(w, q) = ||w - q||_2^2$, but we stress that other measures can be considered as well.

The infimum of the rate-distortion objective (2.1) $\forall \lambda$ is referred to as the rate-distortion function in the source coding literature (Wiegand and Schwarz, 2011). It represents the fundamental bound on the performance of lossy source coding algorithms. However, as we have already discussed above, finding the most optimal code that follows the rate-distortion function is most often NP-hard, and can be calculated only for very few types and/or special cases of input sources. Therefore, in practice, we usually relax the problem until we formalize an objective that can be solved in a feasible manner.

Firstly, we fix the binarization map $B$ by selecting a particular (universal) lossless code and condition the minimization of (2.1) on it. That is, now we only ask for the quantizer $Q$, along with its reconstruction values $Q^{-1}$, that minimize the respective rate-distortion objective. Secondly, we will always assume that we encode an $n$-long 1-dimensional random process. Then, objective (2.1) simplifies to:

Given a lossless code $(B, B^{-1})$, find $(Q, Q^{-1})^*$ that

$$
(Q, Q^{-1})^* = \arg\min_{(Q, Q^{-1})} \mathbb{E}_{p(w)} \left[ D(w_j, q_j) + \lambda L_Q(b_j) \right], \quad (2.6)
$$

$\forall j \in \{0, ..., n - 1\}$, where $q_i \in Q_s := \{q_0, q_1, ..., q_{K-1}\} \subset \mathbb{R}$ and $K = |Q_s| < |\mathbb{W}_s| = n$.

For instance, if we choose $B$ such that it assigns a binary representation of fixed-length to all $w_j$, then the minimizer of (2.6) can be found by applying the k-Means algorithm.

The minimizers of (2.6) are called scalar quantizers, since they measure the distortion independently for each input sample. In contrast, vector quantizers measure the distortion in the respective vector space by grouping a sequence of input samples together. It is well known that the infimum of scalar codes are fundamentally more redundant than vector quantizers, however, due to the associated complexity of vector quantizers it is more common to apply scalar quantizers in practice. Moreover, the inherent redundancy of scalar quantizers is negligible for most practical applications (Wiegand and Schwarz, 2011).

We stress that although the distortion in (2.6) is measured independently for each sample $w_j$, the binarization $b_j$ (and consequently the respective code-length) can still depend on the other samples by taking correlations into account.

Scalar Lloyd algorithm

An example of an algorithm that finds a local optimum is the Lloyd algorithm. It approximates the average code-length of the quantized samples $q_i = (Q^{-1} \circ Q)(w_j)$ with the entropy of their empirical probability mass distribution (EPMD), which we denote throughout this thesis as the first-order entropy. Thus, it substitutes the code-length in (2.6) by $L_C(b_j) = -\log_2 P_{\text{EPMD}}(q_j)$ and applies a greedy algorithm in order to find the most optimal quantizer $Q$ and quantization points $Q^{-1}$ that minimize the respective objective. We show its pseudocode in the appendix B.
CABAC-based RD-quantization

If we are given a set of quantization points $Q_s$ and select CABAC as our universal lossless code, then we can trivially minimize (2.6) by sequentially quantizing the input samples. In the video coding standards the set of quantization points are predefined by the particular choice of quantization strength $\lambda$ (Sze, Budagavi, and Sullivan, 2014). However, in the context of neural network compression we do not know of a good relationship between the quantization strength and the set of quantization points. In the next chapter 3 we describe how we tackled this problem.

2.4 Relation between source coding and model compression

In the last section we have reviewed some fundamental results of source coding theory, which describe the trade-offs between the achievable compression ratios and distortions induced by it. However, in this work we are rather interested in the general topic of model compression. In particular, in the context of DNNs we are interested in compressing the information entailed in their parameter set, under the constrain that the resulting prediction performance is minimally affected by it. Thus, there is a subtle difference between both paradigms. However, they are related under the Minimum Description Length principle (MDL), which we will formally describe in the following for the unsupervised learning setting.

2.4.1 The Minimum Description Length principle

Assume an unsupervised learning setting where given a particular dataset, $D = \{x | x \in \mathcal{X}\}$, we want to learn to predict the elements $x$ as good as possible. The MDL principle equates the task of “learning” with the task of compression. That is, the more regularities we find in the data, the more we can use these to compress it and ultimately, generalize better to unseen data that entail the same regularities. Therefore, the above unsupervised learning goal can be restated as to find the lossless compression algorithm or code that compresses the most the set of data points $x$ in $D$.

Due to the fundamental correspondence between probability distributions and bit-lengths (Grünwald and Rissanen, 2007), we can precisely formalize this idea. Namely, let $\mathcal{W} = \{p(\mathcal{X} | W) | W \in \mathbb{R}^n, n \in \mathbb{N}\}$ be a set of parametric conditional probability distributions over the set $\mathcal{X}$, where we denote as $W \equiv p(\mathcal{X} | W)$ a particular model or point hypothesis in $\mathcal{W}$ (e.g., a trained neural network). Then, one possible way to encode the data points in $D$ is by using a so called two-part code. That is, we first describe a particular point hypothesis $W$ in $\mathcal{W}$ using $L_W(W)$ bits and then optimally encode its prediction errors using $L(\mathcal{X} | W) = - \log_2 p(\mathcal{X} | W)$ bits. Hence, the MDL principle states that we should find,

$$W^* = \min_W \left( \underbrace{- \log_2 p(\mathcal{X} | W)}_{\text{encoding prediction error}} + \alpha \underbrace{L_W(W)}_{\text{encoding model}} \right)$$

(2.7)

where $0 < \alpha \in \mathbb{R}$.

Notice, that (2.7) aims to minimize the prediction errors of the model and the explicit bit-length of the model. Therefore, the MDL principle is a natural framework for the task of model compression.

Thus, under the MDL paradigm,

---

2We remark that the problem can be analogously formulated for other learning tasks.
source coding and machine learning can be interpreted as to be two sides of the
same coin, where model compression can be interpreted as being a particular
coding mechanism for compressing a given dataset.

Thus, interestingly, the RD-curve can be interpreted as describing the minimum
amount of information that a ML algorithm must learn in order to attain a partic-
ular prediction error rate.

2.4.2 DNN model compression

In the following we formulate the model compression problem more specifically to
DNNs. In the context of DNN models we assume that we are given only one (DNN)
model sample with \( n \) real-valued parameters, that is, the parameter set contains \( n \) el-
ements. Thus, here the parameter set to be coded is equivalent to the ones discussed
in the above section 2.3. In addition, we assume a universal coding setting, where
the decoder has no prior knowledge regarding the distribution of the parameter val-
ues. We argue that this simulates most real-world scenarios.

Let now \((x, y) \in \mathcal{D}\) be a set of data samples related to, e.g., a particular super-
vised learning task. Let further \( y' \sim P(y'|x, w) \) denote a trained neural network
model with parameters \( w \) on the dataset \( \mathcal{D} \). Finally, let \( B \) be a chosen and fixed
universal lossless code. Then, we aim to find a quantizer \( Q^* \) that minimizes

\[
(Q, Q^{-1})^* = \arg \min_{(Q, Q^{-1})} \sum_{(x, y) \in \mathcal{D}} \mathcal{L}(y, y'') + \lambda L_Q(b) \tag{2.8}
\]

with \( y'' \sim P(y''|x, q) \) being outputs of the quantized model \( q = (Q^{-1} \circ Q)(w) \) and \( b = (B \circ Q)(w) \).

Analogously to (2.7), the first term in (2.8) expresses the minimization of the
usual learning task of interest, whereas the second term explicitly expresses the
code-length of the model. That is, in the case of DNNs, a code is now applied to
the parameter set of the model which outputs a compressed binary representation,
measures its length in bits, and outputs a set of quantized/reconstructed values.
We stress that the quantized parameter set may be different from its original values.
Hence, following the MDL paradigm, objective (2.8) states that we aim to find the
code that minimizes the bit-length of the parameter set, however, under the con-
strain that the prediction performance of the quantized version of it is minimally
affected.

However, finding the minimum of (2.8) is also most often NP-hard. This moti-
vates further approximations where, as a result, one can directly apply techniques
from the source coding literature in order to minimize the desired objective.

2.4.3 Relaxation of the DNN model compression problem into a “classi-
cal” source coding problem

We now may further assume that the given unquantized model has been pre-trained
on the desired task and that it reaches satisfactory accuracies. Then, it is reason-
able to replace the first term in (2.8) by the KL-Divergence between the unquantized
model \( P(y'|x, w) \) and the respective quantized model \( P(y''|x, q) \). That is, now we aim
to quantize our model such that its output distribution does not differ too much
from its original version.
2.5 Methods & techniques that reduce the complexity of DNNs

Furthermore, if we assume that the output distributions do not differ too much from each other, then we can approximate the KL-Divergence with the Fisher Information Matrix (FIM). Concretely,

$$
E_{P_D}[D_{KL}(y'' || y')] = \delta w F \delta w^T + O(\delta w^2)
$$

(2.9)

with $\delta w = q - w$ and

$$
F := E_{P_D}E_{P(y'|x,w)}\left[\partial_w \log P(y'|x,w)(\partial_w \log P(y'|x,w))^T\right]
$$

Then, by substituting (2.9) in (2.8) we get the following minimization objective

$$
(Q, Q^{-1})^* = \min_{(Q, Q^{-1})} (q - w)F(q - w)^T + \lambda L_Q(b)
$$

(2.10)

Objective (2.10) now follows the same paradigm as the usual source coding problem. However, with the peculiarity that now $D(w, q)$ (approximately) measures the distortion of $w$ and $q$ in the space of output distributions instead the euclidian space. The advantage of the rate-distortion objective (2.10) is that, after the FIM has been calculated, it can be solved by applying common techniques from the source coding literature, such as the scalar Lloyd algorithm.

However, minimizing (2.10) as well as estimating the FIM for deep neural networks usually requires considerable computational resources, and is most often infeasible in practical scenarios. Therefore, we usually consider only the diagonal elements of the FIM (FIM-diagonals), which can be efficiently estimated (see appendix B). As a result, (2.10) simplifies to

$$
(Q, Q^{-1})^* = \arg \min_{(Q, Q^{-1})} F_i(q_i - w_i)^2 + \lambda L_Q(b)
$$

(2.11)

$\forall i \in \{0, \ldots, n - 1\}$, which can be feasibly solved.

Figure 2.6 sketches the difference between the “classical” source coding and model compression paradigms when applied to DNNs.

2.5 Methods & techniques that reduce the complexity of DNNs

In the previous subsections we gave a brief introduction to basic concepts from source coding and data compression. We also explained conceptually how compression of the parameter set may lead to increase in execution efficiency. Now we will introduce some of the concrete optimization techniques that have been studied in the literature.

As figure 2.7 shows, we can optimize the efficiency of DNNs on three fronts:

1. the topology,
2. the parameter set, and
3. the hardware or end-device which processes the information entailed in both components.

In the following we will summarize some of the recent methods & techniques that concerned the optimization of processing DNNs from those three fronts. However, to the interested reader, we refer to (Sze et al., 2020; Deng et al., 2020) which
give an excellent overview on the wide spectrum of related research corpora on this topic.

2.5.1 Optimizations involving the DNNs topology

As figure 2.7 shows, by optimizing the topology of the DNNs we also optimize the processing of the parameter set. This is trivially accomplished by, for instance, selecting smaller or less amount of layers. Moreover changing the topology can also improve the pipelining of the elements of the parameter set on the target hardware platform, consequently reducing its resource consumption. In the following we briefly summarize some of the techniques that are applied nowadays that include optimization of the DNNs topology.

\[
min_{(Q, Q^{-1})} D(w_j, q_j) + \lambda L(b)
\]

\[
arg \min_{(Q, Q^{-1})} L(y, y'') + \lambda L(b)
\]
2.5. Methods & techniques that reduce the complexity of DNNs

Figure 2.7: Different, conceptually high-level components involved in the processing of DNNs that affect their final resource consumption. The DNNs topology affects the parameter set, since it defines its size, shape, etc. The information entailed in the parameter set, as well as in the topology itself, is directly processed by the hardware or end-device where the DNN model is executed. The hardware responsible for processing this information is the sole cause of the resource consumption (e.g. energy, time, material, economical, etc.).

Neural architecture search

Neural architecture search (NAS) is concerned with the problem of finding the most optimal neural network topology for a particular task, in an automated fashion. This is a very broad and active topic of research, and as mentioned before, its specifics are out of the scope of this thesis.

However, NAS-based approaches have also been studied as a means for reducing the complexity of DNN models, and are becoming increasingly popular over the recent years. Some of the most popular approaches include constraints that include information regarding the hardware-resources into the search function, so that the NAS algorithm is forced to not only find a GR that solves well the task at hand, but is also efficient when executed on the hardware platform. For instance, recent advances in finding efficient mobile architectures for image classification were possible thanks to NAS-based approaches (Howard et al., 2019), outputting models that are 3% more accurate and $\times 1.17$ faster than its predecessors. Interestingly, these methods are able to find vastly different topologies dependent on which hardware platform they have been trained for. This fact stresses the importance of selecting the right GR for increasing execution efficiency of DNNs.

The main caveat of NAS-based approaches is their usually large resource requirements for training. They most often require tens of thousands of GPU/hours and may take up to several months to finish. Hence, in the attempt to mitigate this issue, current research on NAS methods focus on finding fast search algorithms in order to mitigate this problem. However, this is still a very active topic of research, and we refer to the excellent survey (Elsken, Metzen, and Hutter, 2019) for the interested reader.
Distilling

Another approach related to finding more compact GRs are methods based on distilling (Hinton, Vinyals, and Dean, 2015). These methods are based on a teacher-student paradigm, where a smaller and more efficient student model tries to “distill” the knowledge of a bigger, with more predictive performance, teacher model. This method has shown to attain good performances and is being widely applied by the community. However, the practical caveat of distilling-based approaches are similar to those of NAS, since the most optimal topology of the student model is unknown a priori. Moreover, the predictive performance of the student network is naturally conditioned on the teachers performance, and as of today it is still an open question how to extract and incorporate the teachers knowledge into the students network in the most effective manner.

DNN compilers

Due to the rapid development of new DL models as well as the recent surge of DL frameworks and specialized hardware accelerators, there has been an increasing demand in building DNN-specific compilers. As the authors of (Li et al., 2021) stated, DL compilers take the model definitions described in the DL frameworks as inputs (that is, in their GR), and generate efficient code implementations on various DL hardware as outputs. The transformation between model definition and specific code implementation are highly optimized targeting the model specification and hardware architecture. Specifically, they incorporate DL oriented optimizations such as layer and operator fusion, which enables highly efficient code generation.

DL compilers can help to greatly alleviate the burden of optimizing the DL models on different hardware platforms manually. Moreover, these methods have the advantage that they do not require to apply expensive training procedure and that they are lossless, meaning, that the models do not suffer of any (or only negligible) prediction performance degradation. However, at the same time the attainable efficiency improvements are more limited since compilers have a much more constrained optimization space.

For the interested reader we refer to (Li et al., 2021), which gives an excellent survey regarding current DL compiler technologies.

2.5.2 Optimizing the parameter set

Another set of methods focused on reducing the complexity of processing the parameter set. Over the past years there has been a plethora of work published on this topic, and a great number of different approaches have been proposed. Again, we refer to (Deng et al., 2020) for a comprehensive survey over this topic.

Arguably some of the most popular methods for reducing the complexity of the parameter set are (but are not limited to) sparsification and quantization methods.

Sparsity

Sparsification of the parameter set is the process of increasing the number of 0-valued elements entailed in it. In other words, sparsification is equivalent to increasing the $L_0$-norm of the parameter set. In this context, the sparsity ratio is a commonly used metric to measure the degree of sparsity, which is defined as the ratio between the number of 0-valued elements to the total number of parameters,
Figure 2.8: Sketch of different sparsity strategies. Unstructured sparsity removes connections from the network without taking correlations between them into account. Block-wise sparsity removes block-like structures from the model’s layers, whereas structured sparsity remove entire network structures such as neurons or filters, thus automatically inducing a change in the network topology.

thus \( p_0 = \# 0s / \# \text{total parameters} \). The advantages of having a high sparsity ratio are two-fold:

1. **Size reduction:** By increasing the \( L_0 \)-norm we are implicitly reducing the information entailed in the parameters, and consequently making them more compressible. For instance, one may achieve compression by neglecting the storage of 0-valued elements and, instead, storing only the non-zero values, along with their positions (more on this in chapters 3 & 4).

2. **Computational cost reduction:** The cost of performing computations such as additions or multiplications can be greatly reduced by, for instance, skipping them entirely whenever a 0-valued element is involved, since the output of such operations is trivially predictable (i.e. \( x \times 0 = 0 \) & \( x + 0 = x \)).

It is important to distinguish between what is called unstructured and structured sparsification methods. Unstructured sparsification methods increase the sparsity ratio without following any preferred positional pattern or structure. In other words, there is seemingly no correlation between the positions of the 0 elements in the network. In contrast, structured sparsification methods take strict correlations between the positions of the zero elements into account. Sometimes, structured sparsity is also referred to as pruning in the literature, the distinction being that pruning methods zero-out well-defined macro structures from the model’s topology such as neurons or filters. Figure 2.8 displays the difference between both approaches.

Which type of sparsity method is applied depends on the properties that come along with each of them. In general, with unstructured sparsity one can attain higher sparsity ratios than with structured sparsity methods at same levels of prediction performance (Zhu and Gupta, 2018). However, the resulting unstructured nature of the model’s connectivity is accompanied with a series of challenges. Firstly, compression can be limited since correlations between the positioning of the zero elements can only be taken into account in a limited manner (see chapter 4 for more information regarding how correlations between the parameter elements can be taken into account in order to achieve higher compression ratios). In the worst case, additional information must be stored signaling the positioning of the zero (or equivalently non-zero) elements. Secondly, conventional general-purpose hardware platforms such as CPUs and GPUs are not well designed for efficiently exploiting the unstructured nature of the positioning of the zero elements. This is due to the inherent
difficulty in identifying the number of non-zero elements that need to be processed at any given moment. This renders inefficient control flows, forcing one to either sacrifice parallel processing, or incurring idle processing units or forcing the processing of redundant, zero elements. Moreover, the unpredictability also difficulties exploiting data locality. In contrast, structured sparsity can overcome the above problems since the correlation between the positioning of the zero elements is well defined a priori. However, as mentioned above, enforcing correlations is a harder constraint which usually comes at the expense of incurring higher drops in prediction performance. Therefore, finding the right trade-off between the prediction performance, sparsity ratio and structurness is essential for attaining a highly efficient DNN model, which is still a current active topic of research.

For the interested reader, we refer to (Blalock et al., 2020) who gives a good overview of most recent and popular sparsification approaches in the literature.

Quantization

A different technique often applied to reduce the complexity of the parameter set is quantization. As explained in section 2.3, quantization consists of mapping the parameter set values into a finite set of smaller cardinality, consequently losing information in the process. Similarly to sparsity, the benefits from applying quantization are two-fold:

1. **Size reduction:** We can attain compression since each parameter value may be represented with a lower amount of bits. Concretely, if $K \in \mathbb{N}$ is the cardinality of the quantized set (that is, the amount of quantization points between the min and max values), then we need $\max b = \log_2 K$ bits in order to represent each element in the set. Thus, as long as $K < 2^{b_{\text{org}}}$ with $b_{\text{org}}$ being the bit-width used to represent the original values of the parameter set, then we automatically attain compression after quantization is applied to it.

2. **Computational cost reduction:** We can reduce the cost of computations per parameter element since less amount of bits need to be processed for each operation. E.g. performing multiplications with 8bit values costs less amount of resources than 32bit values. Moreover, certain type of quantizations allow for more efficient processing on a hardware level. For instance, an 8bit floating point multiplication can consume x3 more energy than an 8bit fixed-point multiplication (Horowitz, 2014).

One of the most common quantization methods employed in the DNN literature is uniform quantization. It consists of mapping the parameter values into a set of quantization points which are equidistantly separated, thus forming a uniform grid in the real line. By constraining the quantization points in such a manner we allow for fixed-point representation and computations of the parameters, thus attaining compression and an increase in computation efficiency of the model simultaneously. However, other quantization methods with similar benefits were also studied in the literature. Again, we refer to (Sze et al., 2020; Deng et al., 2020) as comprehensive overview of the most recent and popular quantization methods studied in the DNN literature.

Lossless Compression

To recall, in the domain of lossless NN compression we are given an already quantized model and the goal is to apply an universal lossless code to its parameter set.
2.5. Methods & techniques that reduce the complexity of DNNs

in order to maximally compress it. Since one of the main contributions of this thesis relates to this topic, in the following we briefly introduce some of the most popular lossless compression techniques applied in the literature.

It is important to distinguish between succinct and non-succinct compact representations. Succinct representations allow to perform certain queries in their compressed representation. In the case of lossless DNN compression, succinct representations permits to perform inference while maintaining their compact form, thus without having to go through an, perhaps expensive decoding process a priori.

**Succinct representations:** As mentioned above, a trivial lossless compression method is to reduce the bit-width of the parameter values according to the cardinality of the set of quantization point. If the quantized elements can be represented in binary representations that are compatible with the numerical formats understood by the computation engine, such as floating point or fixed-point formats, then the compact representation of the parameters is automatically succinct. However, this usually results in a highly redundant model representation, greatly limiting the attainable compression ratios.

Another line of work apply compressed tensor formats such as Compressed Sparse Row (CSR) representation. These tensor data structures do not only offer higher compression gains, but also an associated dot product algorithm that does not require decoding and which minimizes the number of operations needed to execute it. However, these matrix representations are also redundant in that they do not approach the reachable, first-order entropy limit (2.3) (section 2.3.1). To recall, we define the first-order entropy as the entropy of the empirical probability mass distribution of the parameter set. (Han, Mao, and Dally, 2016) attempted to extract some of the redundancies entailed in the CSR representations by applying a scalar Huffman code to its numerical arrays. However, as explained in 2.3.1, this has again the same fundamental limitations that come by applying the scalar Huffman code, one of them being that decoding is required for inference, thus rendering the representation as not succinct.

**Non-succinct representations:** Even though decoding is necessary for inference, applying entropy coders to the models parameter set may still be beneficial in many real-world applications (e.g., in order to reduce parameter size on disk or in federated learning scenarios where parameters are send through a communication channel with limited capacity). Moreover, the inference complexity may still be reduced if the right trade-off between decoding complexity and resource savings achieved by compression is set.

Although lossless compression of neural networks parameters is entirely equivalent to the usual lossless source coding setting discussed in sections 2.3.1 and 2.3.2, there has been relatively little work studying the impact of applying state-of-the-art entropy coding techniques to the networks parameters. In fact, to the best of our knowledge, only the scalar Huffman code and the bzip2 entropy coder belong to the most popular entropy coders applied to the parameter set in the literature (Han, Mao, and Dally, 2016; Choi, El-Khamy, and Lee, 2017; Choi, El-Khamy, and Lee, 2020). However, as we have already discussed in section 2.3.1, these codes have several disadvantages compared to other state-of-the-art lossless codes such as arithmetic codes. Probably the most prominent one is that the scalar Huffman code is sub-optimal in that it incurs up to 1 bit of redundancy per parameter being encoded. This can be quite significant for large networks with millions of parameters. For
instance, VGG16 (Simonyan and Zisserman, 2015) contains 138 million parameters, meaning that the binary representation of any quantized version of it may have up to \(17\text{MB}\) of redundancy if we encode it using the scalar Huffman code.

### 2.5.3 Specialized hardware

As figure 2.7 shows, the last component which is explicitly responsible for the resource consumption of executing DNN models is the hardware component. As briefly explained in the previous section 2.2, data movement between hardware components is the source of mayor resource consumption. Read & write operations from on-chip memory units is up to one order of magnitude more expensive than performing computations, whereas it can be up to three orders of magnitude more costly when it is from off-chip memory units. Therefore, it may not come to a surprise that in recent years substantial research efforts have focused on minimizing the data movement for performing inference. Many incorporate architectural techniques in order to maximize data reuse, by leveraging on unique properties that are characteristic to DNN inference. For instance, many apply spatial reuse techniques in order to increase the data reuse efficiency of convolutional neural networks. However, most often hardware optimizations are tailored to a particular data movement paradigm. Some of the most popular include

- **Weight stationary.** This optimization is tailored to minimize the data movement of the parameters of the network.
- **Output stationary.** Minimizes the data movement of the output values of the computations.
- **Input stationary.** Minimizes the data movement of the input activation values of the computations.
- **Row stationary.** Optimizes the data movement of the elements involved in performing the scalar product.

In parallel, one can take hardware design considerations that reduce the distance between frequently used data, as such reducing further its resource consumption. Others design their hardware so that it can leverage sparsity and/or quantization (that is, reduced precision of the parameter set) in order to increase the data movement as well as the computational efficiency. These approaches can reach highest efficiency performances, specially in terms of energy consumption and area costs, but require preprocessing of the DNN architectures as explained above.

Here we give only a brief overview of different hardware design considerations that one can make for optimizing the inference of DNNs. However, we refer again to (Sze et al., 2020; Deng et al., 2020) for a more detailed overview of the different hardware optimization techniques studied in the literature.
Chapter 3

Reducing the information content on DNNs weights

As described in the previous chapters, it is desired from a learning as well as resource-efficiency perspective to reduce the amount of information entailed in the parameter set of DNN models. Hence, in this chapter we introduce a novel mechanism for doing so. We firstly derive a measure of their explicit information content by expressing it in terms of the first-order entropy. Subsequently, we derive continuous relaxations of it, allowing us to minimize it during training by applying scalable, gradient-based optimization techniques.

We stress once more that in this chapter we do not consider changes in the models architecture, nor optimizations of it. Thus, we only focus on compressing their parameter set.

3.1 Entropy-constrained regularization of DNN training

To recall, one of our goals is to derive an expression for the explicit information content of a DNN parameter set. We start by formalising the general learning problem under the MDL principle since, as we briefly discussed in section 2.4 in chapter 2, the objective of model compression arises naturally under this paradigm.

Assume a supervised learning setting where given a particular data set, \( D_N = \{(x_i, y_i) | x_i \in X, y_i \in Y, i \in \{1, ..., N = |X| = |Y|\}\} \), we want to learn to predict the elements \( y_i \) of the output (or label) set \( Y \) from the elements \( x_i \) of the input set \( X \). Let again \( \mathcal{W} = \{ p(Y|X, W) | W \in \mathbb{R}^n, n \in \mathbb{N} \} \) be a set of parametric conditional probability distributions over the label set \( Y \), where we denote as \( W \equiv p(Y|X, W) \) a particular model or point hypothesis in \( W \) (e.g., a trained neural network). Then, the MDL principle states that we should find,

\[
W^* = \min_W - \log_2 p(Y|X, W) + \alpha L_W(W)
\]

(3.1)

where \( 0 < \alpha \in \mathbb{R} \).

That is, (3.1) aims to minimize the prediction errors of the model and the explicit bit-length of the model simultaneously. Now we need to define a description or code that assigns an unique bit-string to each model \( W \).

Firstly, given a particular model topology which is fixed, we consider the entire set of discrete parameter tensors \( W \). That is, the elements of the parameter set are

\[D_3\]

Disclaimer: Major portions of the content of this chapter are reprinted and with permission from (Wiedemann et al., 2019), ©2020 IEEE.
restricted to be elements from a finite set of real numbers. Thus, \( w_i \) denotes the \( i \)-th weight element of the neural network, which can take one of \( K \) possible values, i.e., \( w_i \equiv \omega_i \in \Omega = \{\omega_0, ..., \omega_{K-1}\} \). We furthermore assume that the network has \( n \) parameters, i.e., \( i \in \{1, ..., n\} \).

Secondly, we assume that the decoder\(^1\) has no prior knowledge regarding the correlations between the input and output set. Consequently, it can neither make any reasonable a priori assumptions regarding the correlations between the weight element values of the neural network. This scenario is very common in most real world cases. For instance, most digital processor units implicitly make such assumption by not prioritising any particular network configuration. Moreover, this is also a common assumption in machine learning problems.

Thus, the a reasonable prior over the weight elements is an \( n \)-long discrete and independent random process. That is, let \( p_{\Omega} = \{p_0, ..., p_{K-1}\} \) be a particular probability mass distribution (PMD) over the finite set \( \Omega \). Then, we model the joint probability distribution as \( P(W) = P(w_1 = \omega_{k_1}, ..., w_n = \omega_{k_n}) = \prod^n_i p_{k_i} \), where \( p_{k_i} \in P_{\Omega} \) is the probability of the \( i \)-th weight element outputting the value \( \omega_{k_i} \in \Omega \). Hence, the respective bit-length can be defined as \( L_W(W) = -\log_2 P(W) = \sum^n_i -\log_2 p_{k_i} \). However, the resulting code depends on the choice of the PMD, which we do not know a priori neither.

A reasonable choice of PMD in this case would be to assume a uniform prior over \( \Omega \), i.e., \( p_k = 1/K \). However, it assigns a bit-string of constant size to each weight element and consequently to each model \( W \). Concretely, \( L_W(W) = n \log_2 K \), and if \( K = 2^{32} \) we obtain the amount of bits needed to store the network if we store each weight element in, e.g., its respective single precision floating point representation. In this case, the minimization (3.1) becomes the standard maximum likelihood estimation objective and consequently no model compression is performed in the process.

However, we can design more suitable codes for the case of deep neural networks. Namely, for each particular model \( W \), we could first estimate the probability distributions of its weight values and subsequently use these estimates in the compression process. This may work well since the number of parameters in most state-of-the-art deep neural network models is typically \( n = \mathcal{O}(10^7) \) and therefore, we can expect the estimation to be a good approximation of the real distribution due to the law of large numbers.

### 3.1.1 Universal coding of discrete deep neural networks

Hence, we propose to first calculate the maximum likelihood estimate of the PMD of \( W \), or in other words its empirical probability mass distribution (EPMD), and then optimally encode the element values of \( W \) relative to it. Such coding techniques belong to the subject of universal coding and their properties are well studied in the literature (Rissanen, 1978; Barron, Rissanen, and Yu, 2006; Xie and Barron, 2000; Grünwald and Rissanen, 2007). Concretely, from (Grünwald and Rissanen, 2007) we know that if we apply an universal two-part code for the task probability density estimation by using histograms, the respective regret (or maximum redundancy) per data point decreases sublinearly as the number of data points increases. That is, the maximum redundancy of the code decreases at a rate of order \( \mathcal{O}(\log_2 n) \), \( n \) being the number of data points. This means that we can expect the regret to be almost 0

---

\(^1\)The entity which is able to recover back the label set \( \mathcal{Y} \) from its compressed bitstream representation.
3.1. Entropy-constrained regularization of DNN training

![Diagram](image)

Figure 3.1: The task of compression is analogous to the task of communication, which is depicted in this figure. Alice wants to send to Bob a particular output set \( Y \). Assume that Alice and Bob have previously agreed on the input set \( X \) and a particular neural network topology. Then, Alice will send \( Y \) performing three steps: 1) she trains a particular network model \( p(Y|X,W) \equiv W \), encodes the unique elements \( \Omega = \{3.2, 1.9, 0.7\} \) (blue, green, red) that appear in the network using an uniform code, along with their empirical probability mass distribution (EPMD) \( \mu = (3/8, 3/8, 2/8) \), and sends these to Bob. 2) She encodes the weight values using an entropy-coder with regards to \( \mu \) and sends them to Bob. Since Bob also knows \( \mu \) and \( \Omega \), he can uniquely recover the weight values. 3) Alice encodes the prediction errors of \( W \) using \( -\log_2 p(Y|X,W) \) bits and sends these to Bob. With that, Bob can uniquely reconstruct the output set \( Y \). ©2020 IEEE.

if we apply such code to the parameter set of DNNs and consequently, the EPMD will almost be identical to the actual PMD that generated the weight values (again, assuming that such a distribution actually generated the weight values).

Hence, we propose the following coding scheme: Let \( \mu = \{\hat{p}_0, ..., \hat{p}_{K-1}\} \) denote the EPMD of the elements \( \Omega \) in \( W \), thus, \( \hat{p}_k = \#(\omega_k)/n \) where \( \#(\cdot) \) denotes the counting operator. Then, we

1. calculate \( \mu \) and encode its elements along with \( \Omega \) using an uniform code, and
2. compress each \( w_i \) using \( -\log_2 \sum_k l_{ik} \mu_k \) bits respectively.

Here, \( l_{ik} \) denotes the indicator function, being 1 if \( w_i = \omega_k \), and 0 otherwise. Fig. 3.1 sketches this coding scheme.

Thus, the resulting bit-length of a particular neural network model \( W \) can be written as,

\[
L_W(W) = L(\|W\|) + L(\mu) + L(\Omega)
\]  

(3.2)

where \( L(\|W\|) = \sum_{i=1}^{n} -\log_2 \sum_k l_{ik} \mu_k \), \( L(\mu) = K \log_2 n \), and \( L(\Omega) = Kb \) with \( b \) being the bit-precision used to represent the elements in \( \Omega \).

Notice how the first term in equation (3.2) can be rewritten as to expressing the first-order entropy of the parameters. That is, \( L(\|W\|) = n \sum_{k=1}^{K} -\mu_k \log_2 \mu_k = nH(\mu) \), with \( H(\mu) \) being the entropy of the EPMD \( \mu \). Thus, (3.2) states that the bit-length of each weight element \( w_i \) is equal to its minimum average bit-length relative to \( \mu \), plus additional redundant terms that result from having to send the estimates. Here we can clearly see that the average redundancy induced by having to send the estimates...
Chapter 3. Reducing the information content on DNNs weights

decreases sublinearly as the number of parameters \( n \) increases, since the second and third term decrease at a rate \( O\left(\frac{\log n}{n}\right) \) & \( O\left(\frac{1}{n}\right) \) respectively when we divide equation (3.2) by \( n \).

### 3.1.2 Entropy-constrained minimization objective

Hence, since the bit-length of the estimation \( L(\mu) \) and \( L(\Omega) \) is constant and the contribution of the second and third term in equation (3.2) to the size of the DNN parameter set can be neglected for large \( n \), the MDL principle states that we ought to find the model that minimizes the objective

\[
W^* = \min_{W \in W} - \log_2 p(Y|X, W) + \alpha n H(\mu), \quad 0 < \alpha \in \mathbb{R}
\]  

(3.3)

The advantages of explicitly minimizing the first-order entropy of the parameter set of a model are two-fold:

1. **Size reduction:** Firstly, notice that the above description of the code-length of a DNN model explicitly expresses the amount of bits required to store its parameter set. Thus, it explicitly upper bounds the total amount of information needed to physically store the model since, as we discussed in section 2.3, we can compress the parameter set to that size if we would apply, e.g., a simple arithmetic coder.

2. **Computational cost reduction:** Secondly, by minimizing the first-order entropy we encourage sparsity as well as quantization sets \( \Omega \) with low cardinality. As we explained in section 2.5.2, by doing so we can achieve higher computational efficiencies since operations with zero elements have trivial outputs and computations with lower bit-widths can be executed more efficiently on a hardware level. However, lowering the entropy of the parameter set induces an additional statistical property which can be exploited in order to attain further computational efficiency gains. Namely, low entropy statistics encourages a high degree of repetition of few unique values which, as we thoroughly describe in chapters 5 and 6, can be exploited in order to minimize the total number of operations that need to be performed for inference and, with it, its dynamic power consumption.

Albeit these advantages, notice that the minimization objective (3.3) is discrete since we assumed that the elements of the parameter set belong to a discrete set \( \Omega \). Thus, we cannot trivially minimized it by applying gradient-based minimization techniques such as stochastic gradient descent (SGD). However, we would like to apply gradient-based optimization techniques since they are scalable and have shown to be capable of finding good local minima that generalize well (Montavon, Orr, and Müller, 2012; LeCun, Bengio, and Hinton, 2015). Hence, in the following sections we describe methods that allow to either implicitly or explicitly minimize (3.3) by gradient-based techniques.

### 3.2 Variational formulation of the entropy-constrained minimization objective

In this section we firstly derive a continuous relaxation of (3.3), and subsequently reformulate it as a variational minimization objective. Then, we will conjecture that,
under certain conditions, the variational objective upper bounds the discrete one and consequently minimizes (3.3).

### 3.2. Continuous relaxation

Notice that the only reason (3.3) is non-differentiable is due to the indicator operator implicitly entailed in it. Hence, we propose to smooth it by a continuous parametric probability distribution. Namely,

\[
I(i = k) = I_{ik} = \begin{cases} 
1, & \text{if } w_i = \omega_k \\
0, & \text{else} 
\end{cases}
\]

\[
\text{cont. relax. } P_i = \left\{ P(i = k|\theta_{ik}) \mid \sum_{k=1}^{K} P(i = k|\theta_{ik}) = 1 \right\}
\]  

(3.4)

with \( P(i = k|\theta_{ik}) = P_{ik} \) being the probability that the weight element \( w_i \) takes the discrete value \( \omega_k \), parametrized by \( \theta_{ik} \). Thus, the continuous relaxation of the indicator function models the probability of an element \( w_i \) taking a particular element of the discrete set \( \Omega \), instead of modeling a concrete selection of a particular element in it, consequently “smoothing” its one-hot “decission”.

Subsequently, we can now naturally replace each estimate \( \mu_k \) of the PMD, and thus the entropy, by

\[
\mu_k = \frac{1}{n} \sum_{i=1}^{n} I_{ik} \quad \text{cont. relax. } P_k = \frac{1}{n} \sum_{i=1}^{n} P_{ik}
\]

\[
H(\mu) = \sum_{k=1}^{K} -\mu_k \log_2 \mu_k \to H(P) = \sum_{k=1}^{K} -P_k \log_2 P_k
\]

(3.5)

Thus, now we have an expression of the entropy \( H(P) \) which is differentiable with regards to the parameters \( \theta_{ik} \), as long as all the \( P_{ik} \) are parametrized through a continuous differentiable function.

Hence, due to reasons that will become more clear later, now we reformulate (3.3) as the following variational minimization objective

\[
\theta^* = \min_{\theta} E_{P_{\theta}} [-\log_2 p(Y|X, W \sim P_{\theta})] + \alpha n H(P)
\]

(3.6)

Thus, we now aim to minimize the averaged prediction error of the network, as taken relative to our parametric probabilistic model \( P_{\theta} \), constrained by the respective relaxation of the entropy (3.5).

However, we can still not apply gradient-based optimization techniques in order to minimize (3.6) since we do not have a differentiable expression for the average log-likelihood (that is, the first term in (3.6)) with regards to the parameters \( \theta \). Moreover, calculating the mean of the log-likelihood is most often infeasible for deep neural networks, so that further techniques need to be considered.

### 3.2.2 Training neural networks under the entropy-constrained variational objective

In order to train the network under the objective (3.6) we apply similar scalable approximation techniques as proposed by the bayesian neural networks literature (Kingma, Salimans, and Welling, 2015; Achille and Soatto, 2018; Molchanov, Ashukha,
Algorithm 1 Forward pass sampling from a discrete probability distribution of the layer’s weight tensor.

```plaintext
1: procedure FORWARDPASS(a) ▷ per layer
2: Calculate the mean weight tensor \( v_W \) and its respective variance \( \sigma^2_W \), as parametrized by the distribution \( P_\theta \) and the set \( \Omega \).
3: Forward pass layer’s mean \( v_z \), and standard deviation \( \sigma_z \), as specified in (3.7) and (3.8).
4: Sample a tensor \( \epsilon \sim \mathcal{N}(0,1) \) with the same dimension as the preactivation layers.
5: Calculate the preactivation values as \( Z = v_z + \sigma_z \circ \epsilon \) ▷ \( \circ \) denotes the Hadamard product.
```

and Vetrov, 2017; Louizos, Ullrich, and Welling, 2017). Concretely, we approximate the mean of the log-likelihood by an unbiased Monte-Carlo estimator and sample from the preactivation values of the network, whose sufficient statistics depend on the mean and variances of the weights and input activations of the respective layer. That is, each preactivation value is now modeled as

\[
Z = v_z + \sigma_z \mathcal{N}(0,1)
\]

\[
v_z = v_W \cdot a
\]

\[
\sigma_z = \sqrt{\sigma^2_W \cdot a^2}
\]

and \((v_W)_i = \sum_k \omega_k P_{ik}, (\sigma^2_W)^{2}_i = \sum_k \omega^2_k P_{ik} - (v_W)_i\) are the mean and variances of the weights of the respective layer, and \(a\) are the activation values of the previous layer.

From a source coding point of view this procedure simulates a stochastic quantization scheme of the weights relative to the joint probability model \( P_\theta \), which can be expressed as Gaussian noise in the preactivation layers due to the central limit theorem. The respective pseudocode can be seen in algorithm 1. Notice, that now (3.6) is differentiable with respect to the parameters \( \theta \) and discrete values \( \Omega \).

After training, we then select the most likely point hypothesis as our discrete model. Henceforth, we will refer to this operation as the maximum-a-posteriori (MAP) quantization step, and denote it as \( W^* = q(\theta^*) \). If all \( P_{ik} \) are defined in terms of a radial function relative to a continuous parameter \( \theta_i \) and \( \omega_k \), then the MAP quantization corresponds to a nearest-neighbor quantization scheme of the parameter \( \theta_i \) relative to the discrete set \( \Omega \) (as depicted in Fig. 3.2).

### 3.2.3 Relation between the variational and discrete objectives

The motivation behind minimizing (3.6) is not only restricted to the scalability property. Namely, under local convexity assumptions of the variational loss landscape, we state in the following theorem that the variational objective upper bounds the desired, discrete objective (3.3).
3.2. Variational formulation of the entropy-constrained minimization objective

Figure 3.2: Quantization of the weight values corresponds to the maximum-a-posteriori (MAP) operator applied to their parametric discrete probability distribution. In our experiments we parametrized the probability distributions using Gaussian kernels, whose MAP operator is equivalent to a nearest-neighbor quantization scheme, as depicted in the above diagram. For instance, all continuous weight element values (green triangles) that lie in region 1 will be quantized to the maximum probable value of that region, namely the value $\omega_1 = -0.4$ (red dot). Analogously for the regions 2 and 3. For constant variance, the decision thresholds (red lines) lie exactly in the middle between two elements of $\Omega = \{-0.4, 0, 0.6\}$. ©2020 IEEE.

**Theorem 1 (Loss upper bound).** Let the probability $P_{ik}$ of a weight parameter $w_i$ sampling the discrete value $\omega_k$ be defined by the Gaussian kernel as follows

$$P_{ik} = \frac{G_\sigma(w_i, \omega_k)}{Z_i},$$

with $G_\sigma(w_i, \omega_k) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{w_i - \omega_k}{\sigma}\right)^2}$

and $Z_i = \sum_k G_\sigma(w_i, \omega_k)$

Let further the probability $P_k$, and the respective entropy $H(P)$ be defined as in (3.5). Finally, let $\theta = (\hat{W}, \Omega, \sigma)$ denote the set of parameters parametrizing the probability distributions $P = \{P_k | k \in \{0, ..., K-1\}\}$, with $\theta^*$ minimizing the variational objective (3.6), and with $W^* = q(\theta^*), \mu^*$, being the respective MAP point hypothesis and respective EPMD. Then, for sufficiently small variance of the distribution $P_{\theta^*}$, and assuming that the loss function is locally convex around the minima, the following bound holds

$$\mathcal{L}(P_{\theta^*}) = \mathbb{E}_{P_{\theta^*}}[-\log_2 p(Y|X, W \sim P_{\theta^*})] + \alpha n H(P(\theta^*))$$

$$> \mathcal{L}(W^*) = -\log_2 p(Y|X, W^*) + \alpha n H(\mu^*)$$

(3.9)

We provide the proof of theorem 1 in the appendix A.

3.2.4 Experiments

We conducted a series of experiments to validate our assumptions and test the effectiveness of compressing DNNs under the variational entropy-constrained loss function (3.6).
Experimental setup

In our experiments, we chose to parametrize each $P_{ik}$ with the Gaussian kernel as stated in theorem 1.

We also chose a different set of discrete values $\Omega^l$ for each layer $l$, where we denote their respective cardinality as $|\Omega^l| = K_l \in \mathbb{N}$. Henceforth, we introduce the notation $\bar{K} = [K_0, K_1, ..., K_l, ..., K_{L-1}]$ in order to specify the cardinality of each layer. Since each layer has a different set of discrete values, we calculated the overall entropy of the network as the sum of the entropies of each individual weight element.

The experiments were performed on the MNIST dataset, using the LeNet-300-100 and LeNet-5 networks, and on the CIFAR-10 dataset, using a VGG16 network like architecture (with 15 layers), referred to as VGG-Cifar-10\(^2\). These networks are commonly used to benchmark different compression methods (Han, Mao, and Dally, 2016; Molchanov, Ashukha, and Vetrov, 2017; Louizos, Ullrich, and Welling, 2017; Ullrich, Meeds, and Welling, 2017). We investigated two compression approaches:

- **Direct entropy-constrained training.** Starting from a pre-trained model, we minimized the Shannon entropy of the network parameters with the variational entropy-constrained objective (3.6) (by applying algorithm 1 to each layer). We used ADAM (Kingma and Ba, 2015) as the optimizer with a linearly decaying learning rate. We also linearly increased (from 0 to up to a number less than 1) the regularization coefficient $\alpha$ in (3.6). Afterwards, in the evaluation stage, the network weights were quantized using the values of the vector $\Omega^l$, available in the $l$-th layer. We set their respective initial cardinalities, $\bar{K}$, as follows:

  - LeNet-300-100 $\bar{K} = [3, 3, 33]$,
  - LeNet-5 $\bar{K} = [5, 5, 33]$, and
  - VGG-Cifar-10 $\bar{K} = [33, 17, 15, 11, 7, 7, 7, 5, 5, 3, 3, 3, 17, 33]$.

  The quantized network performance was monitored with the classification error, compression ratio, and sparsity (calculated on the weights).

- **Pre-sparsifying the models.** Although the entropy term encourages sparsity, in order to better guide the optimization path we simplify the problem by firstly pre-sparsifying the network and subsequently further minimizing their entropy. We applied the sparse variational dropout technique (Molchanov, Ashukha, and Vetrov, 2017) for the sparsification process. To this end, the network parameters were initialized from scratch and ADAM was applied as the optimizer with a linearly decaying learning rate. Also, we linearly increased the regularizer of the Kullback-Leibler divergence approximation during the sparsification stage (from 0 up to a number less than 1). After sparsification, we applied our entropy-constrained training approach to the neural networks. In this context, $\bar{K}$ was defined for each network as:

  - LeNet-300-100 $\bar{K} = [21, 21, 31]$,
  - LeNet-5, $\bar{K} = [17, 17, 31]$, and
  - VGG-Cifar-10, $\bar{K} = [27, ..., 27, 33]$.

\(^2\)http://torch.ch/blog/2015/07/30/cifar.html
3.2. Variational formulation of the entropy-constrained minimization objective

Figure 3.3: Probability mass distribution (PMD) of the weight element values of the last layer of the LeNet-5 model after training. The PMD of the Q-LeNet-5 (S+ECO) is very sparse. In contrast, the PMD of the trained Q-LeNet-5 (ECO) architecture is neither sparse nor the cardinality of its values is particularly low. ©2020 IEEE.

Notation remark: Henceforth, we refer to C-model-name or Q-model-name to either the continuous or quantized model. Furthermore, ECO refers to models that have been trained directly under the entropy-constrained objective, whereas with S+ECO we refer to models that have been a priori sparsified. For instance, Q-LeNet-5 (S+ECO) refers to the quantized LeNet-5 model, which has firstly been sparsified and then trained under the entropy-constrained objective.

Generalizing sparsity and cardinality reduction

Figure 3.3 shows PMDs of the last fully connected layer of the Q-LeNet-5 (ECO) and Q-LeNet-5 (S+ECO) models after training. Interestingly, directly minimizing the entropy-constrained objective admits solutions where the resulting weight distribution is not necessarily sparse, neither its cardinality is particularly lower than at the starting point, thus generalizing the set of possible solutions. In contrast, by pre-sparisifying the network, we constrain the set of solutions to only those which are close to the spike-and-slab distribution.

Nevertheless, from table 3.1 we see that our entropy term is also strongly encouraging sparsity, since overall we attain similar pruning results as current state-of-the-art pruning techniques. This is due to the fact that by increasing sparsity we also minimize the entropy. Concretely, let $p_0$ denote the probability of 0 being sampled. Then, we can rewrite the entropy of a discrete probability distribution as follows

$$H(P) = - \sum_{k=0}^{K} p_k \log_2 p_k = -p_0 \log_2 p_0 - \sum_{k=1}^{K} (1 - p_0) \hat{p}_k \log_2 (1 - p_0) \hat{p}_k$$

$$= H(p_0) + (1 - p_0) H(\hat{P})$$

with $\hat{P}$ being the probability distribution of the sample set, excluding the 0 element.
Figure 3.4: Trend of the classification loss of the VGG architecture when trained on the CIFAR10 data set. The plot shows how the continuous loss does not upper bound the quantized loss during training. This may explain the loss in accuracy performance in Table 3.1 as compared to previous literature. However, we see a correlation between the variance and the difference on the loss values, in that lower variance significantly reduces the difference (see Fig. A.2). ©2020 IEEE.

That is, we can rewrite the entropy of a discrete probability distribution as the entropy of the 0 element plus the entropy of the distribution taken by the subset of the other elements, weighted by the dual probability of the 0 element. With the above expression of the entropy, we can now clearly see that \( H(P) \xrightarrow{P_0=0} 0 \).

Network compression

Table 3.1 summarizes the achieved compression gains. As we can see, the results are comparable to the state-of-the-art compression techniques, as such validating our approach. Interestingly, pre-sparsifying the VGG network seems to significantly improve its final accuracy. We believe that this is due to the variance of the continuous model, which may be too high during the training procedure. For large deep networks with millions of parameters, such as the VGG model, the probability that the quantized model \( P_1 \) (or close models to it) is sampled during training reduces significantly. Consequently, the variational loss value was not upper bounding the cost of the quantized network during training, as can be seen in figure 3.4. However, the variance of the pre-sparsified VGG network was significantly lower during the entire training procedure (see figure A.2 in appendix), hence, closing the gap between the variational and quantized losses and, consequently, attaining higher accuracies on the quantized model.
3.2. Variational formulation of the entropy-constrained minimization objective

We also compare the results to other compression techniques.

| Model                  | Error [ % ] | $\frac{|W^i|}{|W|}$ [ % ] | CR |
|------------------------|-------------|----------------------------|----|
| **S + ECO**            |             |                            |    |
| LeNet-300-100          | 1.80        | 3.28                       | 92 |
| LeNet-5                | 0.90        | 1.11                       | 209|
| VGG-Cifar-10           | 10.91       | 6.77                       | 71 |
| **ECO**                |             |                            |    |
| LeNet-300-100          | 2.24        | 4.03                       | 102|
| LeNet-5                | 0.97        | 1.94                       | 235|
| VGG-Cifar-10           | 14.27       | 6.90                       | 95 |
| **Deep Compression**   |             |                            |    |
| LeNet-300-100          | 1.58        | 8                          | 40 |
| LeNet-5                | 0.74        | 8                          | 39 |
| **Soft Weight Sharing**|             |                            |    |
| LeNet-300-100          | 1.94        | 4.3                        | 64 |
| LeNet-5                | 0.97        | 0.5                        | 162|
| **Sparse Variational Dropout** | | |    |
| LeNet-300-100          | 1.92        | 1.47                       | 68 |
| LeNet-5                | 0.75        | 0.35                       | 280|
| VGG-Cifar-10           | 7.30        | 1.53                       | 65 |
| **Bayesian Compression**|             |                            |    |
| LeNet-300-100          | 1.80        | 2.20                       | 113|
| LeNet-5                | 1.00        | 0.60                       | 771|
| VGG-Cifar-10           | 9.20        | 5.50                       | 116|

Table 3.1: Compression gains attained from training the networks under the variational entropy-constrained minimization objective.
3.3 Entropy-constrained quantization-aware training: A generalization of the straight-through estimator

In the previous section 3.2 we derived a continuous, variational formulation of the discrete, entropy-constrained objective for training DNNs (3.3). The motivation was to be able to apply gradient-based optimization techniques in order to minimize the loss function in a feasible manner. However, one limitation of this approach is that the variational loss function upper bounds the desired loss function only under certain conditions, which may not always be satisfied during training (as can be seen in figure 3.4). Thus, the training under the variational objective may not always be reliable, in the sense that we risk the possibility of attaining high degradation in predictive performance after applying post-training quantization and compression. Therefore, in this section we introduce a different approach that explicitly minimizes the desired entropy-constrained objective (3.3), thus making the results reliably robust to post-training quantization.

3.3.1 Quantization as a source of noise

As mentioned in the previous section, the problem we face is that the quantization operator is non-differentiable. One way of seeing this is by modelling the quantization error as a source of noise that is added to the input signal. That is,

\[ \hat{x} = Q(x) = x + \epsilon \]

with \( \epsilon \sim X \) being a random variable, whose moments are conditioned to the input sample \( x \). We remark that modelling the quantization operator in this manner is common in the source coding literature, specially when analysing the statistical properties of quantization operators is desired (Schuchman, 1964).

Since the samples \( \epsilon \) are generated by operators where the gradients are not well-defined, such as the hard threshold function (1 if the argument is positive, 0 otherwise), then gradients of the quantization operator \( Q(\cdot) \) are consequently also not well-defined.

3.3.2 The straight-through estimator

One way to circumvent this problem is to apply the so called straight-through estimator (STE). The STE was introduced by (Bengio, Léonard, and Courville, 2013) as an heuristic manner to backpropagate gradients through noisy activation function. The basic idea is to

... simply backpropagate through the hard threshold function as if it had been the identity function.

In other words, simply ignore the added noise factor \( \epsilon \) and backpropagate \( Q(\cdot) \) only through the first term in equation (3.3.1).

Many have applied this idea in order to train DNNs to be robust to post-training quantization, so much so that it is becoming one of the de-facto methods of quantization-aware training (Deng et al., 2020). Conceptually, the proposed training algorithm is as follows:

1. forward pass under the quantized model
2. backward pass with regards to the full-precision model
3.3. Entropy-constrained quantization-aware training: A generalization of the straight-through estimator

Algorithm 2 Entropy-constrained quantization-aware training based on the STE

1: Before training, initialize parameter set $W$ and discrete set of quantization points $\Omega$. Then, at each training iteration:

2: **procedure** TRAINING ITERATION

3: For all layers, quantize their parameter tensors $W_q = Q(W, \Omega)$ by the entropy-constrained Lloyd algorithm (Wiegand and Schwarz, 2011) and replace them as the new parameter set of the model. Store full-precision parameter values $W$ in some placeholder.

4: Forward pass input activations with regards to the quantized parameters $\text{loss} = \text{forward}(a, W_q)$.

5: Replace back the full-precision parameter values $W$ into the layers and backpropagate with regards to $W$.

6: Update $W$.

7: Update each element in $\Omega$ according to their respective set of shared gradients in the model.

3. update full-precision model

4. repeat 1-3 until convergence

The advantage of this approach is that, at each training iteration, we project the error into the domain that satisfies the given constraints, thus driving the optimization path to minima that automatically satisfy them. The limitations is that the computation of the surrogate gradients in this manner may induce high noise in the parameter updates, which can significantly slow down or even inhibit convergence of training. Hence, ML practitioners must carefully choose the correct hyperparameter configuration that best trade-off induced noise vs convergence in order to achieve good results.

3.3.3 Entropy-constraint quantization-aware training

Here we propose an entropy-constrained training algorithm for DNNs based on the STE. It consists of applying the entropy-constraint Lloyd algorithm (Wiegand and Schwarz, 2011) for quantizing the parameter values of the DNN during training. Thus, at each training iteration, we explicitly project and evaluate the loss function as stated in the discrete entropy-constrained objective (3.3).

In addition, in order to further conform with the desired minimization objective (3.3), we propose to simultaneously optimize the elements of the discrete set $\Omega$ with regards to the tasks loss function by fine-tuning them at each iteration. The fine-tuning process consists on updating the elements in $\Omega$ as a function of their respective shared gradients in the model.

A pseudocode of the algorithm is depicted in algorithm 2 and a sketch in figure 3.5. In this sketch we exemplify the method when 4bit precision parameters are selected at initialization point with a generalized basis representation, thus leading to 16 possible quantization points.
3.3.4 Experiments

Notice that, due to Lloyds quantization step, the complexity of algorithm 2 increases with the number of parameters of the model and the size of the finite set \( \Omega \) multiplicatively, thus in the order of \( O(Kn) \) (with \( |\Omega| = K \)). Therefore, in our published work (Marban et al., 2020) we applied this training paradigm to train state-of-the-art DNN models for the special case where \( K = 3 \), that is, for so called ternary DNNs.

Our results show state-of-the-art prediction performances, while attaining better pareto-optimal curves as compared to previous work (see figure 3.6). Moreover, we are able to significantly reduce the memory as well as computational requirements of state-of-the-art architectures. Concretely, we attain \( \times 29 \) reduction in number of parameters on average across different networks and datasets, ranging between \( \times 8 \) to \( \times 60 \). In terms of number of floating point operations, we attained a reduction of \( \times 24 \) on average, ranging between \( \times 5 \) to \( \times 60 \). The accuracy degradation was of \( -1.9\% \) on average, ranging between \( -3.4\% \) to \( -0.9\% \). Table 3.2 summarizes the results, where we show how our compact models are more efficient in terms of required number of parameters and floating point operations for inference as compared to previous state-of-the-art DNN model architectures, specifically designed for solving computer vision tasks with high inference-efficiency.

Our results lead us to be among of the top 5 best submissions in the NeurIPS2019 MicroNet Challenge\(^3\), a competition that had as one of its goals

\[ \text{... incentivizing the development of efficient models and model compression techniques.} \]

\(^3\)https://micronet-challenge.github.io
3.4 Related work

Our work is related to previous work that focused on sparsifying and/or quantizing the models parameters, since constraining the entropy of the models can be seen as a generalization them. To recall, sparsification methods focus on minimizing the $L_0$-norm of the models parameter set, whereas quantization techniques focus on minimizing the cardinality of the set of quantization points $\Omega$. Both minimization objectives are special cases of entropy minimization. That is, low entropy incentivises sparsity as well as the reduction of the cardinality of $\Omega$. However, by regularizing the parameter set by their first-order entropy we can induce both properties simultaneously in a meaningful manner, whilst accepting a broader set of solutions that are neither sparse nor their set of values have low cardinality, but nevertheless still being highly compressible.

Our method was also featured in the competitions keynote speech as one of the “most creative and original” among the submitted proposals. For the interested reader, we refer to (Becking, 2020) for a more thorough description and analysis of our method and the experimental results. The code with further details is publicly available\footnote{https://github.com/d-becking/efficientCNNs}, and we provide a more detailed description of the experimental setup in appendix A as well.

In our follow up work we improved these results even further, by increasing the capacity of the models parameters to 4bit precision and the number of cluster centers to 16, in the same manner as depicted in figure 3.5. We describe in detail the experiments and the results in chapter 6, since the training method was jointly developed with our novel hardware architecture, named FantastIC4.

Figure 3.6: Pareto-optimal fronts of our proposed method (EC2T) (Marban et al., 2020) over Trained-Ternary-Quantization (TTQ) (Zhu et al., 2017). The methods were applied to an image classification network trained on the CIFAR-10 dataset, which followed the ResNet-44 architecture. Every data point in this plot represents a quantized model, trained with a specific level of sparsity, and initialized with different centroid values. In the TTQ approach, the sparsity is controlled via simple thresholding, whereas in the EC2T approach, it is modulated by $\gamma$, which was increased from 0.0 (low sparsity) to 0.4 (high sparsity), in steps of 0.1.
Table 3.2: Summary of compression results from our method EC2T (Marban et al., 2020). # FLOPs denotes number of floating point operations required for performing inference, calculated by adding the number of multiplications and summations together. CR\textsubscript{prms} and CR\textsubscript{FLOP} denote the compression ratio, as taken by the ratio of parameter and FLOPs count respectively. The ratio is calculated with regards to the first model in the respective sub-table, that is CR\textsubscript{prms} = \# parameters\textsubscript{1} / \# parameters\textsubscript{i}, with \# parameters\textsubscript{1} being the number of parameters of the model in the first row and \# parameters\textsubscript{i} of the subsequent ones. CR\textsubscript{FLOP} is calculated analogously. We also average both ratios CR\textsubscript{eff} = 0.5 \ast (CR\textsubscript{prms} + CR\textsubscript{FLOP}) in the last column, in order to show a final score reflecting the potential efficiency gains when considering memory as well as computational savings together.

<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th>Top 1 Acc.</th>
<th># parameters</th>
<th># FLOPs</th>
<th>CR\textsubscript{prms}</th>
<th>CR\textsubscript{FLOP}</th>
<th>CR\textsubscript{eff}</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfficientNet-B1</td>
<td>ImageNet</td>
<td>78.43%</td>
<td>7.72M</td>
<td>1324M</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MobileNet-V2</td>
<td>ImageNet</td>
<td>74.70%</td>
<td>6.90M</td>
<td>585M</td>
<td>1.12</td>
<td>2.26</td>
<td>1.69</td>
</tr>
<tr>
<td>MobileNet-V3</td>
<td>ImageNet</td>
<td>75.20%</td>
<td>5.40M</td>
<td>219M</td>
<td>1.43</td>
<td>6.05</td>
<td>3.74</td>
</tr>
<tr>
<td>EfficientNet (ours)</td>
<td>ImageNet</td>
<td>75.05%</td>
<td>972K</td>
<td>261M</td>
<td>7.94</td>
<td>5.07</td>
<td>6.50</td>
</tr>
<tr>
<td>MicroNet-C100</td>
<td>CIFAR-100</td>
<td>81.47%</td>
<td>8.03M</td>
<td>2487M</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CondenseNet-182</td>
<td>CIFAR-100</td>
<td>81.50%</td>
<td>4.20M</td>
<td>513M</td>
<td>1.91</td>
<td>4.85</td>
<td>3.38</td>
</tr>
<tr>
<td>CondenseNet-86</td>
<td>CIFAR-100</td>
<td>76.36%</td>
<td>520K</td>
<td>65M</td>
<td>15.44</td>
<td>38.26</td>
<td>26.85</td>
</tr>
<tr>
<td>MicroNet-C100 (ours)</td>
<td>CIFAR-100</td>
<td>80.13%</td>
<td>226K</td>
<td>71M</td>
<td>35.53</td>
<td>35.03</td>
<td>35.28</td>
</tr>
<tr>
<td>MicroNet-C10</td>
<td>CIFAR-100</td>
<td>97.02%</td>
<td>8.02M</td>
<td>2487M</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CondenseNet-182</td>
<td>CIFAR-100</td>
<td>96.24%</td>
<td>4.20M</td>
<td>513M</td>
<td>1.91</td>
<td>4.85</td>
<td>3.38</td>
</tr>
<tr>
<td>CondenseNet-86</td>
<td>CIFAR-100</td>
<td>95.00%</td>
<td>520K</td>
<td>65M</td>
<td>15.42</td>
<td>38.26</td>
<td>26.84</td>
</tr>
<tr>
<td>MicroNet-C10 (ours)</td>
<td>CIFAR-100</td>
<td>96.71%</td>
<td>231K</td>
<td>80M</td>
<td>34.72</td>
<td>31.09</td>
<td>32.91</td>
</tr>
<tr>
<td>MicroNet-C10 (ours)</td>
<td>CIFAR-100</td>
<td>95.87%</td>
<td>133K</td>
<td>42M</td>
<td>60.30</td>
<td>59.21</td>
<td>59.75</td>
</tr>
</tbody>
</table>

(Yeom et al., 2021) applies concepts from explainable AI (XAI) in order to identify the least relevant units from the DNN and subsequently remove them. Although they show state-of-the-art pruning results with their method, in its current form it can only be applied for sparsifying/pruning the DNNs parameters.

Other related work have also focused on designing regularizers that lower the implicit information entailed in the networks parameters, based on minimizing a variational lower bound objective (Hinton and Camp, 1993; Kingma, Salimans, and Welling, 2015; Achille and Soatto, 2018). Although minimizing the variational lower bound is also well motivated from a MDL point of view (Honkela and Valpola, 2004), the resulting coding scheme is often impractical for real world scenarios. Therefore, (Ullrich, Meeds, and Welling, 2017; Louizos, Ullrich, and Welling, 2017) focused on designing suitable priors and posteriors that allow to apply practical coding schemes
on the weight parameters after the variational lower bound has been minimized. This includes a final step where a lossless entropy coder is applied to the network’s parameter as proposed by (Han, Mao, and Dally, 2016). Therefore, their proposed framework does only implicitly minimize the bit-lengths of the resulting model. In contrast, our entropy-constrained objective (3.3) explicitly states the bit-size of the resulting network. Moreover, our continuous entropy term (3.5) does not correspond to the minimization of a Kullback-Leibler term, but to the following cross-entropy term instead

\[
nH(P) = KL(q(\theta) || p(\theta)) + H(q(\theta)) = \mathbb{E}_{q(\theta)}[-\log_2 p(\theta)]
\]

with \( q(\theta) = P_\theta = \prod_i P_i \) and \( p(\theta) = \prod_k P_k \), where \( P_i \) and \( P_k \) are defined as in (3.4) and (3.5) respectively. This is again well motivated from a practical coding point of view. Notice how the prior \( p(\theta) \) entails all weight element values in each \( P_k \), and therefore implicitly takes correlations between them into account.

Finally, to the best of our knowledge, a later published work (Oktay et al., 2020) is the only work that also applies an entropy-constraint to minimize the information content of the parameter set. However, motivated by transform coding techniques from the source coding literature (Wiegand and Schwarz, 2011), it generalizes our proposed variational approach and parametrizes the parameter set in a manner that affine transformations of them are allowed. They also proposed a different parametrization of the probability distributions of the parameters, which seems to stabilize the training procedure and lead to better solutions. With their approach they were able to highly compress the models parameters, attaining as of today state-of-the-art compression results. However, applying transform coding techniques to the parameter set may significantly reduce the execution efficiency of the models, since these transformations of the parameter set have to be applied at runtime. In particular, their proposed affine transformations require to firstly decode the entire parameter tensor before it can be applied into the forward pass, which induces a substantial increase in runtime memory requirements that may be prohibitive for several resource-constrained devices.

### 3.5 Summary

In this chapter we introduced the concept of a first-order entropy, which we defined as the entropy of the empirical probability mass distribution of a given sample set. We argued that the first-order entropy of DNN model’s parameters can be considered as a reasonable measure for characterizing their information content. Moreover, we argued that trying to find DNNs that best trade-off the accuracy of the task vs their first-order entropy is not only theoretically well-motivated by the MDL principle, but it also comes with a series of advantages in terms of compression and computational efficiency gains attained when performing inference. We also derived and discussed scalable, gradient-based optimization methods, that allow to regularize the models by their first-order entropy during training.

Our experimental results showed that with our approach we can significantly reduce the entropy of state-of-the-art DNN models, reducing the information entailed in their parameter sets by up to \( \times 63 \) and the number of floating point operations required for inference by \( \times 60 \).


3.6 Limitations

A major practical limitation of the here introduced methods for constraining the entropy of DNNs during training are their computational complexity. In our experience, dividing the learning tasks into two parts attained best results. That is, firstly, we suggest to pre-train the DNN until the desired task accuracy is achieved and, subsequently, perform the entropy-constrained training. However, this procedure requires the computational resources of performing DNN training twice, which may sometimes not be feasible. Moreover, both presented entropy-constrained training methods add hyperparameters to the training procedure (e.g., different learning rates for the parameters $w_i$ vs $\omega_k$), which need to be carefully tuned in order to attain good results. This implicitly adds further training complexity, since it becomes more likely that more experiments need to be run before satisfactory results are attained.

A further limitation is, that minimizing the entropy of the DNN's parameters does not automatically imply a reduction in memory and compute complexity at inference time. It only informs about the potential efficiency gains that can be attained. After all, the parameter sets with low-entropy are still represented in 32bit floating point representation if no post-processing is applied to them after training. Recall, our goal in this thesis is to reduce the memory and computational footprint of DNN models when performing inference.

In the following chapters we will discuss concretely how we can design compression engines that exploit the low-entropy statistics of DNN models in order to output highly compact and computationally efficient representations of them.
Chapter 4

Compact representations of DNNs

In the previous chapter 3 we discussed how we can train DNNs to explicitly minimize the first-order entropy of their parameter set, and thus their information content. However, these methods involve expensive training procedures and access to large amounts of training data, which may be considerable barriers for real-world applications. Moreover, we also did not explain in detail how we can generate binary representations of minimal redundancy after the entropy of the models has been minimized.

For instance, assuming an already quantized model, a relatively trivial compression algorithm would be to follow the paradigm explained in section 3.1.1. That is, we encode the information of the parameter set by applying the following two-pat code:

1. Firstly, a codebook entailing the unique set of values (or quantization points) $\Omega$ that appear in the parameter set is encoded. A standard manner to do so would be to encode a list of float32 values for representing the elements in $\Omega$. The max. amount of bits required for encoding this information would be $32(K + 1)$ bits, with $K = |\Omega|$ (the +1 in comes from having to send first the total size $K$ of the list).

2. Then, the empirical probability mass distribution (EPMD) of the parameter set is calculated and encoded in the bitstream. A relatively trivial manner to do so would be to at first encode the total number of parameters $n$ of the model, and subsequently encode a list of integers expressing the quantity in which the elements in $\Omega$ appear in the parameter set of the model. This information would also require max. $32(K + 1)$ bits of information to be encoded.

3. Lastly, we can encode the entire parameter set into a representation with minimal redundancy with regards to its EPMD by applying an arithmetic coder. As explained in section 2.3.1, the arithmetic coder would output a compressed representation of size $nH(P_{\text{EPMD}}) + 2$, thus very close to the one stated by the first-order entropy.

The total size of the compressed representation of the parameter set would then be $nH(P_{\text{EPMD}}) + 64K + 66$. Thus, the compressed representation would entail minimal redundancies as long as $n \gg K$. This is usually true for large models, but the overhead incurred by having to transmit the quantization points as well as the EPMD can quickly become significant for smaller models employed in, for instance, IoT use-cases. In addition, employing a simple arithmetic coder that operates on the EPMD is not the most efficient in terms of coding throughput, and can consume

Disclaimer: Major portions of the content of this chapter are reprinted and with permission from (Wiedemann et al., 2020a), ©2020 IEEE. Published under the CC BY 4.0 licence.
large amount of resources for the coding procedure. This is in general not desired in practical use-cases, specially if inference in the compressed representation ought to be performed. Lastly, the compression performance of this code is also fundamentally limited by the first-order entropy, since it does not take correlations between the parameter elements into account. However, we can compress the parameter set beyond its first-order entropy if we incorporate a mechanism that is able to capture correlations between the parameters as well.

In this chapter we will introduce a more sophisticated and powerful universal compression algorithm that is applicable to any type of DNN model, overcoming all of the above mentioned drawbacks.

4.1 Introduction

As the size of DNN models grows exponentially over time, there is an ever increasing demand to derive highly compact representations of the models. Again, compression can not only help amortize the costs for storing and communicating large models, but it also potentially increases their execution efficiency. Motivated by this issue, the Moving Picture Expert Group (MPEG) of the International Organization of Standards (ISO) has recently issued a call for a standardization of neural network compression technologies (MPEG Requirements, 2019). The standard attempts to solvent the problem of large sizes involved in the processing of DNN models, and the emerging interoperability issues that arise due to compact forms of DNN models becoming more and more ubiquitous in production-ready environments across industry partners. As a response to the call, major tech companies, including Fraunhofer HHI, have proposed different competing compression technologies. Fraunhofer HHI presented DeepCABAC, a universal compression engine for DNNs that is based on the CABAC engine employed in video compression standards H.264/265. As of today, it is the state-of-the-art compression technology in the literature and industry, and it has been selected as the core technology in the aforementioned standard (Kirchhoffer et al., 2021).

The author of this thesis belonged to one of the main contributors to the design and creation of DeepCABAC. Therefore, the following sections will be dedicated to describing its functionality and properties, and in providing a through benchmark and discussion of its compression results.

4.2 DeepCABAC: A universal compression algorithm for DNNs

In light of the discussions conducted within MPEGs standardization group, we can highlight a set of desiderata that a coder for neural network compression should have.

- **Minimal redundancy**: State-of-the-art deep neural networks usually contain millions of parameters. Thus any type of redundancy in the weight parameters may imply several additional MB being stored. Hence, the code should output a binary representation with minimal redundancy per weight element.

- **Universality**: The code should be applicable to any type of incoming neural networks, without having to know their distribution a priori. Hence, the code should entail a mechanism that allows it to adapt to a rich family of possible parameter distributions.
4.2 DeepCABAC: A universal compression algorithm for DNNs

Firstly, DeepCABAC scans the weight parameters of each layer of the network in row-major order. Then, it selects a particular hyperparameter $\beta$ that will define the set of quantization points. Subsequently, it applies a quantizer on to the weight values that minimizes the respective weighted rate-distortion function (4.1). Then, it compresses the quantized parameters by applying our adapted version of CABAC. Finally, it reconstructs the network and measures the respective accuracy of it. The process is repeated for different hyperparameters $\beta$ until the desired trade-off between accuracy and size of the network is achieved. ©2020 IEEE.

- **High coding efficiency**: The computational complexity of encoding/decoding should be minimal. In particular, the throughput of the decoder should be very high if performing inference on the compressed representation is desired.

- **Configurable error vs. compression strength**: The coder should have a hyperparameter that controls the trade-off between the compression strength and the incurred prediction error.

- **High data efficiency**: It is desirable that the coder requires the least amount of data as possible in order to be applicable.

### 4.2.1 DeepCABAC’s coding procedure

We propose a coding algorithm that satisfies all the above properties. We named it *DeepCABAC*, since its based on applying CABAC on the networks quantized weight parameters. Figure 4.1 shows the respective compression scheme. It performs the following steps:

1. It extracts the parameters of the neural networks layer-by-layer and scans the tensors in row-major order\(^1\).

2. Then, it selects a particular configuration $\beta$ which defines the quantization setting, such as the set of quantization points.

\(^1\)Thus, it assumes a matrix form where the parameters are scanned from left-to-right, top-to-bottom.
3. Subsequently, it quantizes the weight values by minimizing a weighted rate-
distortion function of the form (2.11), which implicitly takes the impact of
quantization on the accuracy of the network into account.

4. Then, it compresses them by applying our adapted version of the CABAC al-
algorithm.

5. Finally, it reconstructs the network and evaluates the prediction performance
of the quantized model.

6. The process is repeated for a set of hyperparameters $\beta$, until the desired accuracy-
vs.-size trade-off is achieved.

The source code for encoding and decoding can be found at https://github.
com/fraunhoferhhi/DeepCABAC.

This approach has several advantageous properties. Firstly, it applies CABAC to
the quantized parameters and therefore we ensure that the code satisfies the desider-
ata 1-3. Secondly, by conducting the compression for a set of hyperparameters for
the quantizer we can select the desired pareto-optimal solutions of the accuracy vs.
bit-size plane, thus satisfying property 4. Finally, since only evaluation of the model
is required in the process, a significantly lower (orders of magnitude less) amount
of data samples are required for the compression process than usually employed for
training.

In the following we will explain in more detail the different components of Deep-
CABAC.

**4.2.2 Lossless coder of DeepCABAC**

Consider the weight distribution of the last fully-connected layer of the trained
VGG16 model displayed in figure 4.2. As we can see, there is one peak near 0
and the distribution is asymmetric and monotonically decreasing on both sides. In our
experience, all layers we have studied so far have weight distributions with similar
properties. Hence, in order to accommodate to this type of distributions, we adopted
the following binarization procedure.

Given a quantized weight tensor in its matrix form, DeepCABAC scans the
weight elements in row-major order and binarizes them as follows:

<table>
<thead>
<tr>
<th>Exp.-Golomb</th>
<th>SigFlag</th>
<th>SignFlag</th>
<th>AbsGr(n)Flags</th>
<th>Unary</th>
<th>FL</th>
</tr>
</thead>
</table>

1. The first bit (or bin), *SigFlag*, determines if the weight element is a significant
   element or not. That is, it indicates if the weight value is 0 or not. This bin is
   then encoded using a binary arithmetic coder, according to its respective con-
   text model (color-coded in grey). The context model is initially set to 0.5 (thus,
   50% probability that a weight element is 0 or not), but will automatically be
   adapted to the local statistics of the weight parameters as DeepCABAC en-
   codes more elements.

2. Then, if the element is not 0, the sign bin or *SignFlag* is analogously encoded,
   according to its respective context model.

---

2For fully-connected layers this is trivial. For convolutional layers we converted them into their
respective matrix form according to (Chetlur et al., 2014).
3. Subsequently, a series of bins are analogously encoded, which determine if the element is greater than 1, 2, ..., $n \in \mathbb{N}$ (hence $AbsGr(n)Flags$). The number $n$ becomes a hyperparameter for the encoder.

4. Finally, the remainder is encoded using an Exponential-Golomb\(^3\) code (Teuhola, 1978), where each bin of the unary part are also encoded relative to their context-models. Only the fixed-length part of the code are not encoded using a context-model (color-coded in blue).

For instance, assume that $n = 1$, then the integer $-4$ would be represented as 111101, or the 7 as 10111010. Figure 4.3 depicts an example scheme of the binarization procedure.

The first three parts of the binarization scheme empower CABAC to adapt its probability estimates to any shape distribution around the value 0 and, therefore, to encode the most frequent values with minimal redundancy. For the remainder values, we opted for the Exponential-Golomb code since it automatically assigns smaller code-lengths to smaller integer values. However, in order to further enhance its adaptability, we also encode its unary part with the help of context models. We left the fixed-length part of the Golomb code without context models, meaning that we approximate the distribution of those values by a uniform distribution (see Fig. 4.2). We argue that this is reasonable since usually the distribution of large numbers become more and more flat and it comes with the direct benefit of increasing the efficiency of the coder.

\(^{3}\)To recall, the Exponential-Golomb code encodes a positive integer $2^k < i \leq 2^{k+1}$ by firstly encoding the exponent $k$ using an unary code and subsequently the remainder $r = i - 2^k$ in fixed-point representation.
Figure 4.3: DeepCABACs binarization of neural networks. It encodes each weight element by performing the following steps: 1) encodes a bit (or bin) named \( \text{SigFlag} \) which determines if the weight is a significant element or not (if its 0 or not). 2) If its not 0, then the sign bin, \( \text{SignFlag} \), is encoded. 3) Subsequently, a series of bins are encoded, which indicate if the weight value is greater equal than 1, 2, ..., \( n \in \mathbb{N} \) (the so called \( \text{AbsGr}(n)Flag \)). 4) Finally, the remainder is encoded. The gray bins (also named regular bins) represent bits that are encoded using an arithmetic coder according to a context model. The other bins, the so called bypass bins, are encoded in fixed-point form. For instance, in the above diagram \( n = 1 \), and thus \( 1 \rightarrow 100 \), \(-4 \rightarrow 111101 \) or \( 7 \rightarrow 10111010 \). ©2020 IEEE.

4.2.3 Lossy coder of DeepCABAC

After establishing CABAC as our choice of universal lossless code, now we aim to find the optimal quantizer that minimizes the objective stated in (2.8) (section 2.4). To recall, this involves the optimization of two components:

- **Assignment**: finding the quantizer \( Q \) that optimally assigns the quantization point (or cluster centre) to each weight parameter,

- **Quant. points**: finding the optimal quantization point values \( q_j = (Q^{-1} \circ Q)(w_j) \).

Since neural networks usually rely on scalable, gradient-based minimization techniques in order to optimize their loss function, finding the quantizers that solve (2.8) becomes infeasible in most cases since \( Q \) is a non-differentiable map. Therefore, we opted for a simpler approach.

Firstly, we decouple the assignment map \( Q \) and the quantization points \( Q^{-1} \) from each other and optimize them independently. The quantization points then become hyperparameters for the quantizer, and their values are selected such that they minimize the loss function directly. This separation between \( Q \) and \( Q^{-1} \) was empirically motivated, since we discovered that the networks performance is significantly more sensitive to the choice of \( Q^{-1} \) than to the assignment \( Q \). We discuss this in more detail in the experimental section.

The quantization points

Since finding the correct map \( Q^{-1} \) for a large number of points can be very complex, we constrain them to be equidistant to each other with a specific step-size \( \Delta \). That is, each point \( q_k \) can be rewritten as to be \( q_k = \Delta I_k \) with \( I_k \in \mathbb{Z} \). This does not only
considerably simplify the problem, but it does also encourage fixed-point representations which can be exploited in order to perform inference with lower complexity\textsuperscript{4}.

The assignment

Hence, the quantizer has two configurable hyperparameters $\beta = (\Delta, \lambda)$, the former defining the set of quantization points and the latter the quantization strength. Once a particular tuple is given, the quantizer $Q_\beta$ will then assign each weight parameter $w_i$ to its corresponding quantization point $q_k$ by minimizing the weighted rate-distortion function

$$Q_\beta(w_i) = k^* = \arg\min_k F_i(w_i - q_k)^2 + \lambda L_{ik}$$

\forall i \in \{0, ..., n - 1\}, where $L_{ik}$ is the code-length of the quantization point $q_k$ at the weight $w_i$ as estimated by CABAC.

As previously mentioned, we perform a grid-search algorithm over the hyperparameters $\Delta$ and $\lambda$ in order to find the quantizer configuration that achieves the desired accuracy vs. bit-size trade-off. However, for that we need to define a predefined set of candidate hyperparameters to look for. In this work we considered two approaches for finding the set of step-sizes, which we denote as DeepCABAC-version1 (DC-v1) and DeepCABAC-version2 (DC-v2).

DeepCABAC-version1 (DC-v1)

In DC-v1 we firstly estimate the diagonals of the FIM by applying scalable bayesian techniques. Concretely, we parametrize the network with a fully-factorized gaussian posterior and minimize the variational objective proposed in (Molchanov, Ashukha, and Vetrov, 2017). As a result, we obtain a mean $\mu_j$ and a standard deviation $\sigma_j$ for each parameter, where the former can be interpreted as its (new) value (thus $w_i \rightarrow \mu_i$) and the latter as a measure of their “robustness” against perturbations. Therefore, we simply replaced $F_i = 1/\sigma_i^2$ in (4.1). This is also well motivated theoretically since (Achille, Rovere, and Soatto, 2019) showed that the variance of the parameters approximate the diagonals of the FIM for a similar variational objective. We also provide a more thorough discussion and a precise connection between them in the appendix.

After the FIM-diagonals have been estimated, we define the set of considered step-sizes as follows:

$$q_k = \Delta I_k, \quad \Delta = \frac{2\|w_{\text{max}}\|}{\sigma_{\text{min}}} + S, \quad S, I_k \in \mathbb{Z}$$

(4.2)

where $\sigma_{\text{min}}$ is the smallest standard deviation and $w_{\text{max}}$ the parameter with highest magnitude value. $S$ is then the quantizers hyperparameter, which controls the “coarseness” of the quantization points. By selecting $\Delta$ in such a manner we ensure that the quantization points lie within the range of the standard deviation of each weight parameter, in particular for values $S \geq 0$. Hence, we selected $S$ to be $S \in \{0, 1, ..., 256\}$.

One advantage of this approach is that we can have one global hyperparameter $S$ for the entire network, but each layer will automatically attain a different value for its step-size if we select one $\sigma_{\text{min}}$ per layer. Thus, with this approach we can adapt

\textsuperscript{4}https://www.tensorflow.org/lite
the step-size to the layers sensitivity to perturbations. Moreover, the quantization will also take the sensitivity of each single parameter into account.

**DeepCABAC-version2 (DC-v2)**

Estimating the diagonals of the FIM can still be computationally expensive since it requires the application of the backpropagation algorithm for several iterations in order to minimize the variational objective. Moreover, it only offers an approximation of the robustness of each parameter, and can therefore sometimes be misleading and limit the potential compression gains that can be achieved. Therefore, due to simplicity and complexity reasons, we also considered to directly try to find a good set of candidate step-size $\Delta \in \{\Delta_0, ..., \Delta_{m-1}\}$. However, due to complexity reasons, we apply the same step-size value to all layers. We do so by applying a first round of the grid-search algorithm while applying a nearest-neighbor quantization scheme (that is, for $\lambda = 0$). This allows us to identify the range of step-sizes that do not considerably harm the networks accuracy when applying the simplest quantization procedure. Then, we quantize the parameters as in eq. (4.1), but without the diagonals of the FIM (thus, $F_j = 1 \forall j$).

Under a limited computational budget, this approach has the advantage that we can directly search for a more optimal set of step-sizes $\Delta$ since we spare the computational complexity of having to estimate the FIM-diagonals. However, since DC-v2 considers only one global step-size for the entire network, it cannot adapt to the different sensitivities of the layers.

### 4.3 Experiments

In this section we benchmark DeepCABAC and compare it to other compression algorithms. We also perform further ablation studies, with the purpose to shed light into how different components involved in DeepCABAC impact the final compression performance.

#### 4.3.1 General compression benchmark

Here we benchmark the end-to-end compression gains attained by applying DeepCABAC. In order to assess its universality, we applied it to a wide set of pretrained network architectures, trained on different data sets. Concretely, we used the VGG16, ResNet50 and MobileNet-v1 architectures, trained on the ImageNet dataset, a smaller version of the VGG16 architecture trained on the CIFAR10 dataset\(^5\), which we denote as Small-VGG16, and the LeNet-300-100 and LeNet5 trained on MNIST.

We compare the two versions of DeepCABAC, DC-v1 and DC-v2, against previously proposed compression schemes. Again, since DeepCABAC is based on quantization and entropy coding and does therefore not rely on retraining, it is important that we do only compare it against compression techniques that fall into the same category. That is, against compression techniques that do only directly apply quantization followed by lossless compression, without relying on any sort of (re)training algorithm (thus, without applying backpropagation) that necessitate access to large amounts of data and hardware resources. Therefore, we benchmark DeepCABAC against the nearest-neighbor quantization scheme with uniformly separated quantization points (a.k.a. uniform quantization) and the weighted-Lloyd algorithm, as

\(^5\)http://torch.ch/blog/2015/07/30/cifar.html
Table 4.1: Compression ratios achieved within an accuracy loss of ± 0.5% from the original accuracy when applying different coding methods. DC-v1 and DC-v2 denote the two versions of DeepCABAC, whereas Lloyd denotes the weighted Lloyd algorithm and uniform the nearest-neighbor quantization scheme with equidistant quantization points. For the latter two, we report the best compression results attained after applying scalar Huffman, CSR-Huffman (Han, Mao, and Dally, 2016) and the bzip2 lossless coding algorithms on to the quantized networks. In parenthesis are the resulting top-1 accuracies and in brackets the sparsity ratios achieved as measured by the number of non-zero parameters divided by the total number of parameters.

<table>
<thead>
<tr>
<th>Models</th>
<th>Org. acc.</th>
<th>Org. size</th>
<th>DC-v1 (Acc. top1 %)</th>
<th>DC-v2 (Acc. top1 %)</th>
<th>Lloyd (Acc. top1 %)</th>
<th>Uniform (Acc. top1 %)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[Spars. %]</td>
<td>top1 %</td>
<td>MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRETRAINED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG16</td>
<td>69.94</td>
<td>553.43</td>
<td>5.84 (69.44)</td>
<td>3.96 (69.54)</td>
<td>7.74 (69.50)</td>
<td>17.37 (69.90)</td>
</tr>
<tr>
<td>ResNet50</td>
<td>74.98</td>
<td>102.23</td>
<td>10.14 (74.40)</td>
<td>10.14 (74.51)</td>
<td>13.04 (74.74)</td>
<td>15.58 (74.64)</td>
</tr>
<tr>
<td>MobileNet-v1</td>
<td>70.69</td>
<td>17.02</td>
<td>21.40 (70.21)</td>
<td>22.08 (70.21)</td>
<td>15.00 (68.10)</td>
<td>24.23 (70.10)</td>
</tr>
<tr>
<td>Small-VGG16</td>
<td>91.54</td>
<td>60.01</td>
<td>6.35 (91.11)</td>
<td>5.88 (91.13)</td>
<td>9.98 (91.59)</td>
<td>16.18 (91.53)</td>
</tr>
<tr>
<td>LeNet5</td>
<td>99.46</td>
<td>1.722</td>
<td>3.77 (99.23)</td>
<td>2.52 (99.12)</td>
<td>3.96 (98.96)</td>
<td>20.60 (94.95)</td>
</tr>
<tr>
<td>LeNet-300-100</td>
<td>98.32</td>
<td>1.066</td>
<td>8.61 (98.04)</td>
<td>5.87 (98.00)</td>
<td>8.07 (97.92)</td>
<td>15.01 (98.30)</td>
</tr>
<tr>
<td>SPARSIFIED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG16</td>
<td>69.43</td>
<td>553.43</td>
<td>1.58 (69.43)</td>
<td>1.67 (69.04)</td>
<td>1.72 (69.01)</td>
<td>2.77 (69.42)</td>
</tr>
<tr>
<td>ResNet50</td>
<td>74.09</td>
<td>102.23</td>
<td>5.45 (73.73)</td>
<td>5.14 (73.65)</td>
<td>5.61 (73.73)</td>
<td>6.68 (73.98)</td>
</tr>
<tr>
<td>MobileNet-v1</td>
<td>66.18</td>
<td>17.02</td>
<td>13.29 (66.01)</td>
<td>12.89 (66.02)</td>
<td>11.16 (65.63)</td>
<td>14.78 (65.71)</td>
</tr>
<tr>
<td>Small-VGG16</td>
<td>91.35</td>
<td>60.01</td>
<td>1.90 (91.03)</td>
<td>1.95 (91.06)</td>
<td>2.08 (91.10)</td>
<td>2.84 (91.20)</td>
</tr>
<tr>
<td>LeNet5</td>
<td>99.22</td>
<td>1.722</td>
<td>0.88 (99.14)</td>
<td>0.87 (99.02)</td>
<td>1.09 (99.25)</td>
<td>3.01 (99.22)</td>
</tr>
<tr>
<td>LeNet-300-100</td>
<td>98.29</td>
<td>1.066</td>
<td>2.26 (98.00)</td>
<td>2.20 (98.00)</td>
<td>1.69 (97.76)</td>
<td>4.17 (98.36)</td>
</tr>
</tbody>
</table>

they are the two most widely applied quantization techniques. Furthermore, we apply the scalar Huffman code, the entropy coder proposed by (Han, Mao, and Dally, 2016) which we denote CSR-Huffman, and the bzip2 entropy coder, as they are as
Chapter 4. Compact representations of DNNs

well some of the most commonly applied lossless compression techniques. See appendix for a more detailed explanation of the respective implementations.

Since the attainable compression gains are highly conditioned by the underlying distribution of the neural networks parameters, we also applied these methods to pre-sparified versions of the above mentioned pretrained models. For that, we employed the variational sparsification algorithm (Molchanov, Ashukha, and Vetrov, 2017) to all networks, except for the VGG16 and ResNet50 due to the high computational complexity demanded by the method. For the latter two, we applied the iterative-pruning approach proposed by (Han et al., 2015). The advantage of employing (Molchanov, Ashukha, and Vetrov, 2017) is that we obtain an estimation of the FIM-diagonals as a byproduct of the methods output, thus being able to directly apply DC-v1 and the weighted Lloyd algorithm after the sparsification process finished. In the former cases (pretrained but non-sparse models), we estimated the FIM-diagonals by minimizing the same variational objective proposed in (Molchanov, Ashukha, and Vetrov, 2017), however, while fixing the parameter values (thus without inducing any sparsity in the process). We would like to refer to the appendix for a more comprehensive explanation on the respective sparsification method as well as a thorough discussion on the estimation of the FIM-diagonals.

Table 4.1 shows the results. It reports the attained compression ratios as calculated by

\[
CR = \frac{\sum_{i=0}^{p} C_s(W_i)}{32n} \times 100\%.
\]

where \( W_i \) is a particular parameter tensor of the network (e.g., 2d convolutional weight tensor), \( C_s(W_i) \) denotes its size in bits (e.g., if \( W_i \) is uncompressed then \( C_s(W_i) = 32n_i \), where \( n_i \) is its total number of elements), \( p \) the total number of parameter tensors present in the network and \( n \) is the total number of parameter elements of the network. Thus, we calculate the ratio between the size of the compressed network (which involves the sum of the compressed and uncompressed parameter tensors) and the original neural network.

As one can see, DeepCABAC is able to attain higher compression gains across all networks (with the exception of the last model) as compared to the previously proposed coders. It is able to compress the pretrained by \textbf{x18.9} and the sparsified models by \textbf{x50.6} on average. In contrast, the Lloyd algorithm compresses the models by \textbf{x13.6} and \textbf{x47.3} on average, whereas uniform quantization only achieves \textbf{x5.7} and \textbf{x25.0} compression gains. In addition, notice how DeepCABAC attains higher compression ratios than state-of-the-art 8-bit quantization methods of MobileNet-v1 (21.40\% < 25\%). In contrast to (Nagel et al., 2019), we attain similar accuracies at higher compression ratios \textit{without applying any a posteriori optimization such as bias correction}. However, we stress that the later technique is orthogonal to DeepCABAC, and it can also be applied in order to further reduce the accuracy loss. Similarly, notice that (Han, Mao, and Dally, 2016) reports a compression ratio of 2.05\% at a top-1 accuracy of 68.83\% of the VGG16 model, whereas we were able to attain a compression ratio of 1.58\% at an accuracy of 69.43\%. To recall, Deep Compression consists on applying the sparsification technique (Han et al., 2015), then the k-Means algorithm followed by the CSR-Huffman entropy coder and, finally, fine-tuning the cluster centers to the loss function. In contrast, we were able to attain higher compression performance at higher accuracies by simply applying (Han et al., 2015) +

\footnote{Although a better compression ratio was attained, we were not able to get an accuracy in the ±0.5 percentage point range of the original accuracy. Therefore, this result shall not be considered as the best result.}
4.3. Experiments

Table 4.2: Comparison of attained compression ratios with related literature. VGG16 results are from (Han, Mao, and Dally, 2016), Mobilenet-v1 from (Nagel et al., 2019), MixNet from 1st place submission to MicroNet-Challenge-19, BERT from (Lan et al., 2020) and YOLO-v3 from (Redmon and Farhadi, 2018). DeepCABAC attains similar (or better) results without the necessity of applying expensive optimizations such as distillation, bias correction or fine-tuning. The top-1 accuracies, F1- and mAP-scores are respectively in parenthesis, and the compression ratios otherwise. All values are measured in percentage.

<table>
<thead>
<tr>
<th>Model</th>
<th>Compression ratios</th>
<th>Related literature</th>
<th>DeepCABAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG16 [553 MB]</td>
<td>2.05 (68.83)</td>
<td>1.58 (69.43)</td>
<td></td>
</tr>
<tr>
<td>MobileNet-v1 [17.02 MB]</td>
<td>25.00 (70.50)</td>
<td>21.40 (70.21)</td>
<td></td>
</tr>
<tr>
<td>MixNet [16.57 MB]</td>
<td>100.00 (75.03)</td>
<td>7.9 (75.03)</td>
<td></td>
</tr>
<tr>
<td>BERT [433 MB]</td>
<td>11.00 (89.00)</td>
<td>9.00 (86.00)</td>
<td></td>
</tr>
<tr>
<td>YOLO-v3 [248 MB]</td>
<td>100.00 (55.30)</td>
<td>8.65 (53.45)</td>
<td></td>
</tr>
</tbody>
</table>

DeepCABAC, without having to perform any a posteriori fine-tuning of the quantization points.

Furthermore, we also applied DeepCABAC to YOLO-v3 (Redmon and Farhadi, 2018) and BERT (Devlin et al., 2019) which are state-of-the-art models for object detection and natural language processing tasks respectively (see table 4.2 and the supplement for an extensive report of the results). In short, these models can be compressed down to 10% of their original size while negligibly affecting their prediction performance. In particular, we were able to attain a compression ratio of 9% at a F1-score of 86% on the BERT model, which is competitive with the reported results from the literature (Lan et al., 2020) (compression ratio of 11% at a F1-score of 89%). However, we remark that (Lan et al., 2020) attained these results after applying expensive retraining, distilling and fine-tuning procedures. In contrast, DeepCABAC achieved the above results by simply applying quantization plus entropy coding techniques.

Finally, we also compressed a sparsified version of MixNet (Tan and Le, 2019b), which is a state-of-the-art model for ImageNet classification designed for efficient use on embedded devices. We were able to compress the network by x12.7 from its original size, attaining as such a network that achieves 75.03% top-1 accuracy and needs only 1.31MB of storage requirements.

We summarise all above results and their comparison with the related literature in table 4.2. They corroborate that DeepCABAC serves as a powerful universal quantizer + entropy coder for neural networks.
Chapter 4. Compact representations of DNNs

Table 4.3: Average bit-sizes per parameter for the Small-VGG16 network after applying different quantizers. DC-v1 and DC-v2 denote the two versions of DeepCABAC, whereas Lloyd denotes the weighted Lloyd algorithm and uniform corresponds to the nearest-neighbor quantization. We chose the networks that resided within the ±0.1 percentage point range from the accuracy attained after applying a uniform quantizer. In the case of the Lloyd and uniform quantizers, the size of the quantized networks were measured with regards to the entropy of their empirical probability mass distribution. In contrast, we measured the explicit average bit-size per parameter in DC-v1 and DC-v2.

<table>
<thead>
<tr>
<th>step-sizes (top1 acc.)</th>
<th>DC-v1</th>
<th>DC-v2</th>
<th>Lloyd</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRETRAINED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.032 (90.35)</td>
<td>1.48</td>
<td>1.48</td>
<td>1.79</td>
<td>1.60</td>
</tr>
<tr>
<td>0.016 (91.13)</td>
<td>2.21</td>
<td>2.20</td>
<td>2.29</td>
<td>2.40</td>
</tr>
<tr>
<td>0.001 (91.55)</td>
<td>4.27</td>
<td>4.80</td>
<td><strong>2.34</strong></td>
<td>5.61</td>
</tr>
<tr>
<td>SPARSIFIED [7.57%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.032 (90.22)</td>
<td>0.47</td>
<td>0.47</td>
<td>0.52</td>
<td>0.48</td>
</tr>
<tr>
<td>0.016 (91.06)</td>
<td>0.59</td>
<td><strong>0.58</strong></td>
<td>0.62</td>
<td>0.60</td>
</tr>
<tr>
<td>0.001 (91.17)</td>
<td>0.91</td>
<td>1.00</td>
<td><strong>0.74</strong></td>
<td>1.00</td>
</tr>
</tbody>
</table>

4.3.2 Ablation study: Assignment vs. quantization points

To recall, lossy quantization involves two types of mappings, the quantization map $Q$ where parameter values are assigned to quantization points (or equivalently integers), and the reconstruction map $Q^{-1}$ which assigns a value to each quantization point. In DeepCABAC the former is influenced by the Lagrangian multiplier $\lambda$ which controls the trade-off between the bit-size and the distortion incurred by the quantization, and the latter by the quantization step-size $\Delta$ which controls the “coarseness” of the quantization points. Both hyperparameters influence the resulting accuracy and compression ratio of the network. Hence, the following experiment aims to assess their respective impact.

Table 4.3 shows the average bitsizes attained after applying different quantizers to the Small-VGG16 network at different step-sizes but fixed accuracies (within a range of ±0.1% from each other). Moreover, the average bitsizes that result from applying the Lloyd and the uniform quantizers are measured in terms of first-order entropy values. To recall, it corresponds to the entropy of the empirical probability mass distribution, which marks the theoretical compression limit for all entropy coders that were considered in conjunction with these quantizers (e.g., scalar Huffman code)$^8$. Thus, the difference in compression performance at a fixed step-size do now only reflect the difference in “assignment-decisions” made by the respective quantizer. In other words, each row in Table 4.3 assess the clustering performance of each quantizer.

There are two main insights we attain from Table 4.2, namely in the regime of strong quantizations (or big step-sizes):

$^7$https://github.com/wps712/MicroNetChallenge

$^8$Other entropy coders that take correlations between the parameters into account, such as CABAC, may attain lower values (as can be seen in Table 4.2 and 4.3).
1. The distortion measures do not seem to reflect the actual accuracy loss that will result from quantization.

2. The simple, nearest-neighbour quantization scheme seems to make (almost) as good assignment decisions as rate-distortion based quantizers.

The first insight may be due to the approximation nature of the distortion measures. Namely, since they do only represent local approximations to the incurred accuracy, they may very well be inaccurate at strong perturbations. Indeed, from Table 4.3 we can also see that as the step-sizes become smaller, rate-distortion based quantizers make better assignment decisions than the uniform quantizers. In particular, we see that DeepCABAC-v1 outperforms DeepCABAC-v2 at smallest step-size, which indicates that the Fisher-weighted rate-distortion cost function serves as a better approximation of the actual MDL-loss function than the non-weighted rate-distortion cost function. Therefore, in the high rate regime (low step-sizes), we can see that the lossy and lossless components are closely coupled together.

The second insight may be considered as a more intriguing result. Namely, we see that in the strong quantization regime (high step-size values) the entropy of the uniform quantization scheme is lower than of the Lloyd quantizer. Moreover, it comes closer to DeepCABAC’s compression performance as we increase the step-size. Thus, the simple nearest-neighbour quantization scheme seems to be a surprisingly good quantizer in terms of compression-vs.-accuracy performance for large quantization step-sizes. Notice, that (Choi, El-Khamy, and Lee, 2017) also reports a similar phenomena. This may be due to rate-distortion quantizers inducing a biased error that affects stronger the accuracy of the network. However, it is not entirely trivial why this is the case, and as of today it still remains an open question.

In conclusion, it seems that the compression performance is more strongly affected by the particular choice of the quantization step-size ∆. These insights motivated the design of DC-v2 in the first place, since it is able to explore a larger set of step-sizes for the best accuracy vs. bit-size trade-offs. Indeed, as Table 4.1 from the previous experiment shows, DC-v2 attains similar or even higher compression gains than DC-v1, in particular in the case of pretrained networks.

### 4.3.3 Ablation study: Lossless coding

In our last experiment we aimed to assess the efficiency of the considered universal lossless coders. To recall, there are two essential components that influence the compression performance of a universal entropy coder:

1. A probability estimate of the data to be coded
2. A mapper that assigns a binary string to the data with minimal redundancy.

For this experiment we quantized the Small-VGG network using three different quantizers, and subsequently compressed each of them using different universal lossless coders. More concretely, we quantized the model by applying DC-v2, the weighted Lloyd algorithm and the nearest-neighbor quantizer. We then applied the scalar Huffman code, the CSR-Huffman code (Han, Mao, and Dally, 2016), the bzip2 algorithm, and the CABAC-component of DeepCABAC. Moreover, we also calculated the first-order entropy of the quantized networks. The resulting average bit-sizes are reported in Table 4.4.

---

9The fact that DeepCABAC performs worse than the Lloyd algorithm for the smaller step-size lies in the algorithmic design choice rather than the rate-distortion decision.
Table 4.4: Compression ratios achieved from lossless compressing different quantized versions of the Small-VGG16 network (and its sparse version). The network was quantized in three different manners, one by applying DC-v2, another with the weighted Lloyd algorithm, and finally with the uniform quantization (nearest-neighbor quantization). The top1 accuracy of each quantized model lies within the ±0.1 percentage point range from the original accuracy of the model, which is 91.54% and 91.35% respectively. Subsequently, each of them was compressed by applying the scalar Huffman code, the CSR-Huffman code (Han, Mao, and Dally, 2016), the bzip2 coder, and by CABAC. The second last row denotes the entropy of the EPMD.

<table>
<thead>
<tr>
<th>Quantizers → Lossless codes ↓</th>
<th>Uniform</th>
<th>Lloyd</th>
<th>DC-v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRETRAINED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scalar-Huffman</td>
<td>5.18</td>
<td>3.19</td>
<td>2.33</td>
</tr>
<tr>
<td>bzip2</td>
<td>5.22</td>
<td>3.22</td>
<td>2.42</td>
</tr>
<tr>
<td>CABAC</td>
<td><strong>4.77</strong></td>
<td><strong>2.74</strong></td>
<td><strong>2.07</strong></td>
</tr>
<tr>
<td>H</td>
<td>5.09</td>
<td>2.91</td>
<td>2.20</td>
</tr>
<tr>
<td>SPARSIFIED [7.57%]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scalar-Huffman</td>
<td>1.35</td>
<td>1.71</td>
<td>1.33</td>
</tr>
<tr>
<td>CSR-Huffman</td>
<td>0.91</td>
<td>0.67</td>
<td>0.65</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.73</td>
<td>0.72</td>
<td>0.71</td>
</tr>
<tr>
<td>CABAC</td>
<td><strong>0.63</strong></td>
<td><strong>0.63</strong></td>
<td><strong>0.61</strong></td>
</tr>
<tr>
<td>H</td>
<td>0.84</td>
<td>0.60</td>
<td>0.58</td>
</tr>
</tbody>
</table>

As one can see, CABAC is able to attain higher compression gains across all quantized versions of the Small-VGG16 network. The benefits from using CABAC come from its inherent flexibility in that it can be accommodated to capture the prior statistics of the weight parameters. Namely, by defining the binarization procedure as in section 4.2.2, DeepCABAC is able to quickly capture the statistics of unimodal and asymmetric distributions, with their maxima close to 0. In addition, we decided for a row-major scanning order so that CABAC is able to capture correlations between elements in a row (which are indeed present as demonstrated in (Nagel et al., 2019)). This is also crucial since CABAC’s estimate is updated in an autoregressive manner and therefore, its compression performance also depends on the scanning order. Indeed, as Table 4.4 shows, CABAC is able to capture correlations between the weight parameters and consequently compress them beyond the first-order entropy of the parameter distribution. This particular property highlights its superiority as compared to the previously proposed universal entropy coders, e.g., scalar Huffman and CSR-Huffman, since their average code-lengths are bounded by the first-order entropy and therefore it would be impossible for them to attain lower code-lengths than CABAC.

Finally, since CABAC applies arithmetic coding, it automatically fulfils point 2) and therefore produces binary representations with minimal redundancies. This allows him to attain average bit-lengths of less than 1 bit per parameter element, which usually reflects the information content in sparse models. In contrast, the average bit-lengths of e.g. the scalar Huffman code is lower bounded by 1 bit (since it incurs at least 1 bit of redundancy per element), thus limiting the attainable compression ratios of the model.
4.4 Follow-up work

DeepCABACs functionality as described in this chapter was up until recently been adopted as the baseline quantization and entropy coding method for the upcoming MPEG-7 standard for compression of neural networks for multimedia content description and analysis (ISO/IEC 15938 part 17) (Video subgroup, 2019). However, further improvements have been made by the authors from the time of publication of DeepCABAC. In particular, in (Haase et al., 2020) we were able to improve the compression performance by 5.8% on average. These gains were accomplished by changing the quantizer into a variation of the Trellis-Coded Quantization employed in the Versatile Video Coding (VVC) standard (Schwarz et al., 2019), which is a structured form of vector quantization. As of today, a combination between DeepCABAC and the improvements described in (Haase et al., 2020) are the core quantization and entropy coding technology for the upcoming MPEG-7 standard (Kirchhoffer et al., 2021).

Moreover, in (Neumann et al., 2020) we show that DeepCABAC can be applied in a “plug & play” fashion in order to also compress significantly updates to the parameter set. Compressing updates of parameters is of high importance since there are several real-world use cases where these have to be transmitted through expensive communication channels. An example may be an update to a mobile app that entails DL models, or “over-the-air” distributed training scenarios such as in Federated Learning (Sattler, Wiegand, and Samek, 2020). Concretely, we were able to show that we can reduce the communication costs involved in sending/receiving parameter updates by x44 when DeepCABAC is applied in a Federated Learning scenario. This work further corroborates the versatility, universality and efficiency of DeepCABAC’s coder.

4.5 Summary

In this chapter we describe a novel compression algorithm for deep neural networks called DeepCABAC. It is based on applying a Context-based Adaptive Binary Arithmetic Coder (CABAC) to the networks parameters, which is the state-of-the-art universal lossless coder employed in the H.264/HEVC and H.265/HEVC video coding standards. DeepCABAC also incorporates a novel quantization scheme that explicitly minimizes the accuracy vs. bit-size trade-off, without relying on expensive retraining or access to large amounts of data. Our experiments showed that we can compress pretrained neural networks by x18.9 on average, and their sparsified versions by x50.6, consistently attaining higher compression performance than previously proposed coding techniques with similar characteristics. Moreover, DeepCABAC is able to capture correlations between the networks parameters, as such being able to compress the networks parameters beyond the first-order entropy limit, surpassing the fundamental limitations of codes that assume a stationary distribution such as the Huffman code.

4.6 Limitations

A major drawback of DeepCABAC’s compressed representations of DNNs is that they cannot be trivially exploited for reducing resources required for performing inference. On the one hand, compression of the weight parameters can greatly help in alleviating the resources spend in loading data from/to memory. Again, loading
data from off- to on-chip can cost up to three orders of magnitudes more resources than performing arithmetic operations. On the other hand, the potential gains are challenged by the additional complexity incurred by processing compressed data. Namely, DeepCABAC’s compressed representation of the weight parameters can not be employed explicitly for performing arithmetic operations. That is, decoding is required beforehand, which adds computational complexity into the inference algorithm. Secondly, the autoregressive nature of CABAC’s coder implies sequential processing of the compressed data, which greatly limits its parallelisation capabilities. In addition, the variable-length nature of the compressed representation greatly complicates the efficient processing of the data, due to the unpredictable irregularities incurred by it.

However, highly efficient implementations CABAC’s decoder are readily available (Marpe and Wiegand, 2003), as well as dedicated decoder chips that are widely adopted by the industry (Sze et al., 2008; Zhou et al., 2012; Richter, Stabernack, and Kühn, 2012; Hahlbeck and Stabernack, 2014). These methods may be employed in order to minimize the complexity incurred by DeepCABAC’s decoder and increase the efficiency of processing compressed representations of data.

Ultimately, the optimal interplay between compression, an efficient implementation of an inference algorithm and the underlying hardware platform, will determine the resource-consumption of inference of a DNN compressed with DeepCABAC. Understanding how to best configure these factors requires a thorough study, which we leave as interesting future work to be pursued.
Chapter 5

Compact and computationally efficient representations of DNNs

In the previous chapter 4 we described DeepCABAC, a coder specially designed for maximally compressing parameter sets of DNN models. Raw compression comes with direct benefits with regards to storage requirements as well as reducing communication costs for sending/receiving parameter sets and their updates. However, DeepCABAC’s compressed representation is not succinct, meaning, that decoding is required prior to performing calculations with the parameter elements. Thus, if inference in the compact representation is desired, then the inference engine would automatically have a higher computational complexity due to the decoding process. This consequently renders a trade-off between the decoders resource requirements and the potential savings attained by compression.

Therefore, in this chapter we study alternative lossless compression algorithms that are succinct, in the sense that they generate compact representations of the models and minimize the computational complexity involved in the inference engine simultaneously.

5.1 Introduction

The dot product operation between matrices constitutes one of the core operations in almost any field in science. Examples are the computation of approximate solutions of complex system behaviors in physics (Landau, Paez, and Bordeianu, 2008), iterative solvers in mathematics (Young and Gregory, 1988) and features in computer vision applications (Krig, 2014). Also deep neural networks heavily rely on dot product operations in their inference (Montavon, Orr, and Müller, 2012); e.g., networks such as VGG-16 require up to 16 dot product operations, which results in 15 giga operations for a single forward pass. Hence, lowering the algorithmic complexity of these operations and thus increasing their efficiency is of major interest for many modern applications. Since the complexity depends on the data structure used for representing the elements of the matrices, a great amount of research has focused on designing data structures and respective algorithms that can perform efficient dot product operations (Afroz et al., 2016; Bläser, 2013; Duff, 1977).

Of particular interest are sparse matrices. To recall, sparsity refers to the property that many of the elements are zero valued. In principle, one can design efficient representations of sparse matrices by only storing the non-zero entries of the matrix, consequently, their storage requirements becoming of the order of the number of non-zero values. However, having a compact representation does not imply

---

Disclaimer: Major portions of the content of this chapter are reprinted and with permission from (Wiedemann, Müller, and Samek, 2020), ©2020 IEEE. Published under the CC BY 4.0 licence.
that the dot product algorithm associated to that data structure will also be efficient. Hence, a great part of the research was focused on the design of data structures that have as well dot product algorithms with low computational complexity (Duff, 1977; Yuster and Zwick, 2004). Moreover, by assuming sparsity alone we are implicitly imposing a spike-and-slab prior\(^1\) over the empirical probability mass distribution (EPMD) of the elements of the matrix. If the actual distribution of the elements greatly differs from this assumption then compressed sparse data structures may entail too many redundancies, as such limiting the attainable compression ratios. Hence, sparsity can be a too constrained assumption for representation of quantized neural networks. Mathematically, sparsity can be considered a subclass of the general family of low entropic distributions. In fact, sparsity measures the min-entropy of the element distribution, which is related to Shannon’s first-order entropy measure through Renyi’s generalized entropy definition (Rényi, 1961).

So, in this chapter we ask the question:

“Can we devise compact and efficient representations under the implicit assumption that the first-order entropy of the distribution of the elements of the parameter tensors of DNN models is low?”

We stress once more that, in this chapter, by efficient we mean in terms of reducing the algorithmic complexity of the dot product associated to the representation. In particular, we focus on the number of elementary operations required in the algorithm, since they correlate with the energy and time complexity when executed on hardware (Horowitz, 2014).

5.2 Data structures for matrices with low entropy statistics

Consider as an example the following matrix

\[
M = \begin{pmatrix}
0 & 3 & 0 & 2 & 4 & 0 & 0 & 2 & 3 & 4 & 0 & 4 \\
4 & 4 & 0 & 0 & 0 & 4 & 0 & 0 & 4 & 4 & 0 & 4 \\
4 & 0 & 3 & 4 & 0 & 0 & 4 & 0 & 2 & 0 & 0 & 0 \\
0 & 0 & 0 & 4 & 4 & 0 & 3 & 4 & 0 & 0 & 0 & 0 \\
0 & 4 & 4 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

5.2.1 Previous compact matrix formats

Dense formats

As we discussed in the previous chapters, in particular in chapter 4, one way to reduce the storage requirements of the matrix \(M\) is to code its elements one-by-one following a particular scanning order, say, row-major order\(^2\) and output a compressed representation of it. For instance, one could apply a codebook-based coder, where the indices pointing to the unique elements present in the matrix are stored and compressed either by reducing their bit-width or by applying an arithmetic coder to them. However, the computational complexity of performing the dot product associated to those representations is at least of the order of the number of elements in the matrix, that is, of \(O(m \times n)\). State-of-the-art DNN models can have parameter tensors entailing up to millions of elements, specially in fully-connected layers.

---

\(^1\)That is, a delta function at 0 and an uniform distribution elsewhere.

\(^2\)left to right, up to down
5.2. Data structures for matrices with low entropy statistics

Therefore, having to process this large number of computations is undesired and can severely compromise the efficiency of the inference engine.

Sparse formats

One way to attain both, reduction in storage requirements and computational complexity is when run-length coding techniques are applied (Wiegand and Schwarz, 2011). The basic principle behind run-length coding techniques is to encode the number of times a particular element appears consecutively, instead of its elements one-by-one.

Sparse matrix formats take advantage of this and encode the zero values in such a manner in order to reduce storage requirements as well as the computational complexity for performing the dot product. Conceptually the principle is simple, namely,

\[ \text{if the matrix entails many 0 elements, then we do not need to store them and, at the same time, we can skip multiplications and additions that involve 0 values.} \]

For instance, the Compressed Sparse Row (or CSR in short) format stores the values of the above matrix \( M \) in the following manner:

1. Scans the non-zero elements in row-major order and stores them in an array. We denote this array as \( W \).
2. Simultaneously, it stores the respective column indices in another array, which we call \( colI \).
3. Finally, it stores pointers that indicate when a new row starts. We denote this array as \( rowPtr \).

Hence, the matrix \( M \) would take the following form

\[
W : [3, 2, 4, 2, 3, 4, 4|4, 4, 4, 4, 4, 4, 4|3, 4, 2|4, 4, 3, 4, 4|4, 4, 4, 4]
\]
\[
colI : [1, 3, 4, 7, 8, 9, 11|0, 1, 5, 8, 9, 11|0, 2, 3, 7, 9|3, 4, 5, 7, 8, 9|1, 2, 5, 7]
\]
\[
rowPtr : [0, 7, 13, 18, 24, 28]
\]

In order to facilitate readability of the above representation, we introduced the delimiter “,” in order to separate each element in the respective arrays, whereas with “|” we indicate the start of a new row. That is, between each “|” are all the non-zero elements and column indices belonging to a particular row of the matrix. Notice, how they are positioned respective to the numbers indicated in \( rowPtr \). We stress that these delimiters are not symbols that need to be stored in the final representation.

If we assume the same bit-size per element for all arrays, then the CSR matrix format does not attain higher compression gains for this particular example (62 entries vs. 60 that are being required by the dense data structure). However, notice how now the storage requirements of the CSR representation is of the order of non-zero elements present in the matrix \( M \). Therefore, if the matrix \( M \) becomes more sparse, then the CSR representation would eventually be smaller than the dense.

Moreover, by taking the CSR format we can reduce the computational complexity for performing the dot product operation. As an example, consider only the scalar product between the second row of matrix \( M \) with an arbitrary input vector...
The dot product algorithm associated with the CSR representation would only multiply-add the non-zero entries by performing the following steps:

1. Load the subset of rowPtr respective to row 2. Thus, rowPtr → [7, 13].

2. Then, load the respective subset of non-zero elements and column indices. Thus, W → [4, 4, 4, 4, 4] and colI → [0, 1, 5, 8, 9, 11].

3. Finally, load the subset of elements of a respective to the loaded subset of column indices and subsequently multiply-add them to the loaded subset of W. Thus, a → [a₀, a₁, a₅, a₈, a₁₀, a₁₁] and calculate 4a₀ + 4a₁ + 4a₅ + 4a₈ + 4a₉ + 4a₁₁.

By executing this algorithm we would require 20 load operations (2 from the rowPtr and 6 for the W, the colI and the input vector respectively), 6 multiplications, 5 additions and 1 write. In total this dot product algorithm requires 32 operations. This is lower than 48 operations (24 load, 12 multiply, 11 add and 1 write) if we would perform the scalar product by naively multiplying and accumulating each element in a sequential manner.

5.2.2 Matrix formats that exploit low first-order entropy statistics

We can improve both results by exploiting the low-entropy property of matrix M. To recall, in chapter 3 we argued that output samples generated by a random process with low first-order entropy exhibit the property that, with a high probability,

\textit{only a few number of unique elements are repeated very frequently and consecutively in the sequence of data samples.}

This is similar to the sparsity property with regard to the zero element. However, now we generalize it and take advantage of the same fact for all unique elements appearing in the matrix. We can do so by applying again run-length coding techniques akin to the sparse formats.

Moreover, similarly to the way sparse formats can naturally skip operations regarding zero values, we can

\textit{leverage the distributive law of multiplications in order to skip multiplications,}

consequently reducing the total number of operations needed to perform the dot product operations.

In the following we propose two new formats which realize this.

\textbf{Compressed Entropy Row (CER) format}

Firstly, notice that many elements in M share the same value. In fact, only the following four values Ω = \{0, 4, 3, 2\} appear in the entire matrix. Secondly, notice that different elements appear more frequent than others, and their relative order does not change throughout the rows of the matrix. Concretely, we have a set of unique elements \( \Omega = \{0, 4, 3, 2\} \) which appear \( P_{\Omega} = \{32, 21, 4, 3\} \) times respectively in the matrix, and we obtain the same relative order of highest to lowest frequent value throughout the rows of the matrix. Hence, we can design a data structure which leverages both properties in the following way:

1. Store unique elements present in the matrix in an array in frequency-major order (that is, from most to least frequent). We name this array \( \Omega \).
2. Store respectively the column indices in row-major order, excluding the first element (thus excluding the indices of the most frequent element). We denote it as \( \text{colI} \).

3. Store pointers that signal when the positions of the next new element in \( \Omega \) starts. We name it \( \Omega \text{Ptr} \). If a particular pointer in \( \Omega \text{Ptr} \) is the same as the previous one, this means that the current element is not present in the matrix and we jump to the next element.

4. Store pointers that signal when a new row starts. We name it \( \text{rowPtr} \). Here, \( \text{rowPtr} \) points to entries in \( \Omega \text{Ptr} \).

Hence, this new data structure represents matrix \( M \) as:

\[
\begin{align*}
\Omega : [0, 4, 3, 2] \\
\text{colI} : [4, 9, 11 : 1, 8 : 3, 7 : 0, 1, 5, 8, 9, 11 : 0, \ldots] \\
\Omega \text{Ptr} : [0, 3, 5, 7 | 13 | 16, 17, 18 | 23, 24 | 28] \\
\text{rowPtr} : [0, 3, 4, 7, 9, 10]
\end{align*}
\]

Here the delimiter “,” also separates each element in the respective arrays and “ | ” indicates the start of a new row. However, in contrast to the CSR format, “ | ” are now positioned inside the \( \Omega \text{Ptr} \) array since \( \text{rowPtr} \) points to entries in it. Additionally, now we introduce the delimiter “:” which indicates the the positions in which a new element appears in the row. That is, between “:” are all column indices that belong to one and only one element in a particular row of the matrix. Notice, how the positions of “:” are set as indicated by \( \Omega \text{Ptr} \). We stress again that these delimiters are not symbols that need to be stored in the final representation.

We refer to this data structure as the \textit{Compressed Entropy Row} (or CER in short) format. One can verify that indeed, the CER format only requires 49 entries (instead of 60 or 62 as for the dense or CSR format respectively) attaining as such a compressed representation of the matrix \( M \).

Moreover, we can design a dot product algorithm associated to the CER format with lower computational complexity than the dense and CSR counterparts. For instance, if we consider again the scalar product between an input vector and the second row of the matrix \( M \), we could attain the same results by perform the following steps:

1. Load the subset of \( \text{rowPtr} \) respective to row 2. Thus, \( \text{rowPtr} \rightarrow [3, 4] \).

2. Load the corresponding subset in \( \Omega \text{Ptr} \). Thus, \( \Omega \text{Ptr} \rightarrow [7, 13] \).

3. For each pair of elements in \( \Omega \text{Ptr} \), load the respective subset in \( \text{colI} \) and the element in \( \Omega \). Thus, \( \Omega \rightarrow [4] \) and \( \text{colI} \rightarrow [0, 1, 5, 8, 9, 11] \).

4. For each loaded subset of \( \text{colI} \), perform the sum of the elements of \( a \) respective to the loaded \( \text{colI} \). Thus, \( a \rightarrow [a_0, a_1, a_5, a_8, a_{10}, a_{11}] \) and do \( a_0 + a_1 + a_5 + a_8 + a_9 + a_{11} = z \).

5. Subsequently, multiply the sum with the respective element in \( \Omega \). Thus, compute \( 4z \).

The operations required by this algorithm are 17 load operations (2 from \( \text{rowPtr} \), 2 from \( \Omega \text{Ptr} \), 1 from \( \Omega \), 6 from \( \text{colI} \) and 6 from \( a \)), 1 multiplication, 5 additions
and 1 write. In total these are 24 operations, which is x2 and x1.3 lower than the dense and sparse algorithms. Again, this gains are attributed to the fact that we leverage the distributive law of multiplications in order to reduce the number of computations needed to perform the scalar product, as perhaps more intuitively displayed in equation form as follows

- **dense**: $4a_1 + 4a_2 + 0a_3 + 0a_4 + 0a_5 + 4a_6 + 0a_7 + 0a_8 + 4a_9 + 4a_{10} + 0a_{11} + 4a_{12}
- **CSR**: $4a_1 + 4a_2 + 4a_6 + 4a_9 + 4a_{10} + 4a_{12}
- **CER**: $4(a_1 + a_2 + a_6 + a_9 + a_{10} + a_{12})$

In the following we will introduce a further data structure which leverages low entropy statistics similarly to CER, but has a slightly more relaxed prior regarding the ordering of the unique elements in the matrix.

**Compressed Shared Elements Row (CSER) format**

In some cases it may well be that the probability distribution across rows in a matrix are not similar to each other. Hence, the second assumption in the CER data structure would not apply and we would only be left with the first one. That is, we only know that not many unique elements appear per row in the matrix or, equivalently, that many elements share the same value. The *compressed shared elements row* (CSER) data structure is a slight extension to the previous CER representation above. Here, we add an element pointer array, which signals which element in $Ω$ the $coll$ indices refer to. We called it $ΩI$. Thus, $ΩI$ points to entries in $Ω$, $ΩPtr$ to entries in $coll$ and $rowPtr$ to entries in $ΩPtr$. Hence, the above matrix would then be represented as follows

- $Ω$ : $[0, 2, 3, 4]$
- $coll$ : $[4, 9, 11 : 1, 8 : 3, 7 : 0, 1, 5, 8, 9, 11 : 0, 3, 7 : 2 : 9 : 3, 4, 5, 8, 9 : 7 : 1, 2, 5, 7]$
- $ΩI$ : $[3, 2, 1, 3, 3, 2, 1, 3, 2, 3]$
- $ΩPtr$ : $[0, 3, 5, 7 | 13 | 16, 17, 18 | 23, 24 | 28]$
- $rowPtr$ : $[0, 3, 4, 7, 9, 10]$

Thus, for storing matrix $M$ we require 59 entries, which is still a gain but not a significant one. The benefits of applying CSER will become more obvious when applied to quantized parameter tensors of DNNs (see experimental section 5.4). Notice, that now the ordering of the elements in $Ω$ is not important anymore, as long as the $ΩI$ array is accordingly adjusted. Similarly, the ordering of $ΩI$ at each row can also be arbitrary, as long as the $ΩPtr$ and $coll$ array are accordingly adjusted. The relationship between CSER, CER and CSR data structures is described in Section 5.3.

A similar dot product algorithm can be devised for the CSER data structure, thus attaining similar gains as the CER format. One can find both pseudocodes in the appendix C.

In the next section we give a detailed analysis about the storage requirements needed by the data structures and also the efficiency of the dot product algorithm associated to them. This will help us identify when one type of data structure will attain higher gains than the others.
5.3 Analysis of the storage and energy complexity of data structures

Without loss of generality, in the following we assume that we aim to encode a particular quantized matrix $M = \omega_k \in \Omega$ take values from a finite set of elements $\Omega = \{\omega_0, \omega_1, ..., \omega_{K-1}\}$. Moreover, we assign to each element $\omega_k$ an empirical probability mass value $p_k = \#(\omega_k) / N$, where $\#(\omega_k)$ counts the number of times the element $\omega_k$ appears in the matrix $M$. We denote as $P_\Omega = \{p_0, p_1, ..., p_{K-1}\}$ the respective EPMD. In addition, we assume that each element in $\Omega$ appears at least once in the matrix (thus, $p_k > 0$ for all $k = 0, ..., K - 1$) and that $\omega_0 = 0$ is the most frequent value in the matrix. Finally, we order the elements in $\Omega$ and $P_\Omega$ in probability-major order, that is, $p_0 \geq p_1 \geq ... \geq p_{K-1}$.

5.3.1 Definition of the computational complexity of the dot product

Since measuring exactly runtime and energy complexity of an algorithm depends on the software implementation and on the hardware the program is running on, we will model the complexity of the dot product in a manner that its agnostic to those factors. Thus, instead we consider the minimum number of elementary operations required to perform the dot product algorithm as a measure of its complexity. That is, we consider:

1. a `mul` or multiply operation which takes two numbers as input and outputs their multiplied value,
2. a `sum` or summation operation which takes two values as input and outputs their sum,
3. a `read` operation which reads a particular number from memory and
4. a `write` operation which writes a value into memory.

Moreover, in order to make our complexity model more flexible with regard to particular software and hardware configurations, we wrap each of those operations with a non-linear function reflecting those dependencies. To this end, we introduce four cost functions $\sigma, \mu, \gamma, \delta : \mathbb{N} \rightarrow \mathbb{R}$, which take as input a bit size and output the cost (e.g. runtime or energy) of performing the operation associated to them$^3$; $\sigma$ is associated to the `sum` operation, $\mu$ to the `mul`, $\gamma$ to the `read` and $\delta$ to the `write` operation.

Note, that we do not consider read/write operations from/into low level memory (like caches and registers) that store temporary runtime values, e.g., outputs from summation and/or multiplications, since their cost can be associated to those operations. Now, each of these nodes can be associated with, e.g., an energy cost as in (Horowitz, 2014). Consequently, the total energy required for a particular dot product algorithm would simply equal the sum of the operations. We want to remark that similar energy models have been previously proposed with the goal of benchmarking the inference complexity of DNNs in an analytical manner (Yang, Chen, and Sze, 2017; Chen et al., 2017).

Considering this measure of algorithmic complexity we can now provide a detailed analysis of the costs involved in the CER and CSER formats. However, we

$^3$The `sum` and `mul` operations take two numbers as input and they may have different bit sizes. Hence in this case, we take the maximum of those as a reference for the bit sizes involved in the operation.
start with a brief analysis of the storage and AC requirements of the dense and sparse data structure in order to facilitate the comparison between them.

5.3.2 Complexity analysis of the dense and CSR formats

The dense data structure stores the matrix in an $N$-long array (where $N = m \times n$) using a constant bit-size $b_\Omega$ for each element. Therefore, its effective per element storage requirements are

$$S_{\text{dense}} = b_\Omega$$  \hspace{1cm} (5.1)

bits. The associated standard scalar product algorithm then has the following per element AC

$$E_{\text{dense}} = \sigma(b_o) + \mu(b_o) + \gamma(b_a) + \gamma(b_\Omega) + \frac{1}{n}\delta(b_o)$$  \hspace{1cm} (5.2)

where $b_a$ denotes the bit size of the elements of the input vector $a \in \mathbb{R}^n$ and $b_o$ the bit size of the elements of the output vector. The cost (5.2) is derived from considering 1) loading the elements of the input vector $[\gamma(b_a)]$, 2) loading the elements of the matrix $[\gamma(b_\Omega)]$, 3) multiplying them $[\mu(b_o)]$, 4) summing the multiplications $[\sigma(b_o)]$, and 5) writing the result $[\delta(b_o)/n]$. We can see that both the storage and the dot product efficiency have a constant cost attached to them, despite the distribution of the elements of the matrix.

In contrast, the CSR data structure requires only

$$S_{\text{CSR}} = (1 - p_0)(b_\Omega + b_I) + \frac{1}{n}b_I$$  \hspace{1cm} (5.3)

effective bits per element in order to represent the matrix, where $b_I$ denotes the bit size of the column indices. This comes from the fact that we need in total $N(1 - p_0)b_\Omega$ bits for representing the non-zero elements of the matrix, $N(1 - p_0)b_I$ bits for their respective column indices and $mb_I$ bits for the row pointers. Moreover, it costs

$$E_{\text{CSR}} = (1 - p_0)(\sigma(b_o) + \mu(b_o) + \gamma(b_a) + \gamma(b_\Omega) + \gamma(b_I))$$

$$+ \frac{1}{n}\gamma(b_I) + \frac{1}{n}\delta(b_o)$$  \hspace{1cm} (5.4)

per matrix element in order to perform the dot product. The expression (5.4) was derived from 1) loading the non-zero element values $[(1 - p_0)\gamma(b_\Omega)]$, their respective indices $[(1 - p_0)\gamma(b_I) + \gamma(b_I)/n]$ and the respective elements of the input vector $[\gamma(b_a)]$, 2) multiplying and summing those elements $[\sigma(b_o) + \mu(b_o)]$ and then 3) writing the result into memory $[\delta(b_o)/n]$.

Different to the dense format, the efficiency of the CSR format increases as $p_0 \to 1$, thus, as the number of zero elements increases. Moreover, if the matrix size is large enough, the storage requirement and the cost of performing a dot product becomes effectively 0 as $p_0 \to 1$.

For the ease of the analysis, we introduce the big $O$ notation for capturing terms that depend on the shape of the matrix. In addition, we denote the following set of operations

$$c_a = \sigma(b_a) + \gamma(b_a) + \gamma(b_I)$$  \hspace{1cm} (5.5)

$$c_\Omega = \gamma(b_I) + \gamma(b_\Omega) + \mu(b_o) + \sigma(b_o) - \sigma(b_o)$$  \hspace{1cm} (5.6)
5.3. Analysis of the storage and energy complexity of data structures

$c_a$ can be interpreted as the total effective cost of involving an element of the input vector in the dot product operation. Analogously can $c_{\Omega}$ be interpreted with regard to the elements of the matrix. Hence, we can rewrite the above equations (5.2) and (5.4) as follows

\[
E_{\text{dense}} = c_a + c_{\Omega} - 2\gamma(b_1) + O(1/n) \\
E_{\text{CSR}} = (1 - p_0)(c_a + c_{\Omega}) + O(1/n)
\]

(5.7) \hspace{1cm} (5.8)

5.3.3 Efficiency analysis of the CER and CSER formats

Following a similar reasoning as above, we can state the following theorem

**Theorem 2.** Let $M \in \mathbb{R}^{m \times n}$ be a matrix. Let further $p_0 \in (0,1)$ be the empirical probability mass distribution of the zero element, and let $b_I \in \mathbb{N}$ be the bit size of the numerical representation of a column or row index in the matrix. Then, the CER representation of $M$ requires

\[
S_{\text{CER}} = (1 - p_0)b_I + \bar{k} + \bar{k}n b_I + O(1/n) + O(1/N)
\]

(5.9)

effective bits per matrix element, where $\bar{k}$ denotes the average number of shared elements that appear per row (excluding the most frequent value), $\bar{k}$ the average number of padded indices per row and $N = m \times n$ the total number of elements of the matrix. Moreover, the effective cost associated to the dot product with an input vector $a \in \mathbb{R}^n$ is

\[
E_{\text{CER}} = (1 - p_0)c_a + \bar{k}n c_{\Omega} + \bar{k}n \gamma(b_1) + O(1/n)
\]

(5.10)

per matrix element, where $c_a$ and $c_{\Omega}$ are as in (5.5) and (5.6).

Analogously, we can state

**Theorem 3.** Let $M$, $p_0$, $b_I$, $\bar{k}$, $c_a$, $c_{\Omega}$ be as in theorem 2. Then, the CSER representation of $M$ requires

\[
S_{\text{CSER}} = (1 - p_0)b_I + 2\bar{k}n b_I + O(1/n) + O(1/N)
\]

(5.11)

effective bits per matrix element, and the per element cost associated to the dot product with an input vector $a \in \mathbb{R}^n$ is

\[
E_{\text{CSER}} = (1 - p_0)c_a + \bar{k}n c_{\Omega} + \bar{k}n \gamma(b_1) + O(1/n)
\]

(5.12)

The proofs of theorems 2 and 3 are trivial in the sense, that they can be directly derived from the definition of their representation and their respective dot product algorithms (see appendix C).

The theorems state that the efficiency of the CER & CSER formats depend on the $(\bar{k}, p_0)$ (average number of unique elements per row & sparsity) values of the empirical distribution of the elements of the matrix. That is, the CER & CSER formats become increasingly efficient for distributions that have high $p_0$ and low $\bar{k}$ values.
However, since the first-order entropy measures the effective average number of unique values that a random variable outputs\(^4\), both values are intrinsically related to it. In fact, from Renyi’s generalized entropy definition (Rényi, 1961) we know that \(p_0 \geq 2^{-H}\). Moreover, the following properties are satisfied

- \(\bar{k} \to \min\{K - 1, n\}\), as \(H \to \log_2 K\) or \(n \to \infty\), and
- \(\bar{k} \to 0\), as \(H \to 0\) or \(n \to 1\).

Consequently, we can state the following corollary

**Corollary 3.1.** For a fixed set size of unique element \(|\Omega| = K\) and constant index bit size \(b_I\), the storage requirements \(S\) as well as the cost of the dot product operation \(E\) of the CER and CSER representations satisfy

\[
S, E \leq O(1 - 2^{-H}) + O(K/n) + O(1/N)
\]

\[
= O(1 - 2^{-H}) + O(1/n)
\]

where \(p_0, b_I, n\) and \(N\) are as in theorems 2 and 3, and \(H\) denotes the entropy of the matrix element distribution.

Thus, the efficiency of the CER and CSER data structures increase as the column size increases, or as the entropy decreases. Interestingly, when \(n \to \infty\) both representations will converge to the same values, thus, will become equivalent. In addition, there will always exist a column size \(n\) where both formats are more efficient than the original dense and sparse representations (see Fig. 5.3 where this trend is demonstrated experimentally).

### 5.3.4 Connection between CSR, CER and CSER

The CSR format is considered to be one of the most general sparse matrix representations, since it makes no further assumptions regarding the empirical distribution of the matrix elements. Consequently, it implicitly assumes a spike-and-slab distribution on them. However, spike-and-slab distributions are a particular class of low entropic (for sufficiently high sparsity levels \(p_0\)) distributions. In fact, spike-and-slab distributions have the highest entropy values compared to all other distributions that have same sparsity level. In contrast, as a consequence of corollary 3.1, the CER and CSER data structures relax this prior and can therefore efficiently represent the entire set of low entropic distributions. Hence, the CSR data structure can be interpreted as a more specialized version of the CER and CSER representations.

This may be more evident via the following example: consider the 1st row of the matrix example from Section 5.2

\[
(0 \ 3 \ 0 \ 2 \ 4 \ 0 \ 0 \ 2 \ 3 \ 4 \ 0 \ 4)
\]

\(^4\)From Shannon’s source coding theorem (Shannon, 1948) we know that the entropy \(H\) of a random variable gives the effective average number of bits that it outputs. Therefore, we may interpret \(2^H\) as the effective average number of distinct elements that a particular random variable outputs.
5.3. Analysis of the storage and energy complexity of data structures

The CSER data structure would represent the above row in the following manner

\[
\Omega : [0, 4, 3, 2] \\
colI : [4, 9, 11, 1, 8, 3, 7] \\
\Omega I : [1, 2, 3] \\
\Omega P tr : [0, 3, 5, 7] \\
\text{rowPtr} : [0, 3]
\]

In comparison, the CER representation assumes that the ordering of the elements in \(\Omega I\) is similar for all rows and therefore, it directly omits this array and implicitly encodes this information in the \(\Omega\) array. Therefore, the CER representation can be interpreted as a more explicit/specialized version of the CSER. The representation would then be

\[
\Omega : [0, 4, 3, 2] \\
colI : [4, 9, 11, 1, 8, 3, 7] \\
\Omega P tr : [0, 3, 5, 7] \\
\text{rowPtr} : [0, 3]
\]

Similarly, the CSR representation omits the \(\Omega P tr\) array since it assumes an uniform distribution over the non-zero elements (thus, over the \(\Omega\) array), and in such case all the entries in \(\Omega P tr\) would redundantly be equal to 1. Therefore, the respective
representation would be

\[ \Omega : [3, 2, 4, 2, 3, 4, 4] \]
\[ \text{colI} : [1, 3, 4, 7, 8, 9, 11] \]
\[ \text{rowPtr} : [0, 7] \]

Consequently, the CER and CSER representations will have superior performance for all those distributions that are not similar to the spike-and-slab distributions. Figure 5.1 displays a sketch of the regions on the entropy-sparsity plane where we expect the different data structures to be more efficient. The sketch shows that the efficiency of sparse data structures is high on the subset of distributions that are close to the right border line of the \((H, p_0)\)-plane, thus, that are close to the family of spike-and-slab distributions. In contrast, dense representations are increasingly efficient for high entropic distributions, hence, in the upper-left region. The CER and CSER data structures would then cover the rest of them. Figure 5.2 confirms this trend experimentally.

5.4 Experiments

We applied the dense, CSR, CER and CSER representations on simulated matrices as well as on quantized neural network weight matrices, and benchmarked their efficiency with regard to the following four criteria:

1. Storage requirements: We calculated the storage requirements according to equations (5.1), (5.3), (5.9) and (5.11).

2. Number of operations: We implemented the dot product algorithms associated to the four above data structures (implementation details and pseudocodes of can be found in the appendix C) and counted the number of elementary operations they require to perform a matrix-vector multiplication.

3. Time complexity: We timed each respective elementary operation and calculated the total time from the sum of those values.

4. Energy complexity: We estimated the respective energy cost by weighting each operation according to Table 5.1. The total energy results consequently from the sum of those values. As for the case of the IO operations (read/write operations), their energy cost depend on the size of the memory the values reside on. Therefore, we calculated the total size of the array where a particular number is entailed and chose the respective maximum energy value. For instance, if a particular column index is stored using a 16 bit representation and the total size of the column index array is 30KB, then the respective read/write energy cost would be 5.0 pJ.

In addition, we used single precision floating point representations for the matrix elements and unsigned integer representations for the index and pointer arrays. For the later, we compressed the index-element-values to their minimum required bit sizes, where we restricted them to be either 8, 16 or 32 bits.

Notice that we do not consider the complexity of converting the dense representation into the different formats in our experiments. This is justified in the context of neural network compression since we can apply this step a priori to the inference procedure. That is, in most real world scenarios one firstly converts the weight matrices, possibly with help of a capable computer, and then deploys the converted
Table 5.1: Energy values (in pJ) of different elementary operations for a 45nm CMOS process (Horowitz, 2014). We set the 8 bit floating point operations to be half the cost of a 16 bit operation, whereas we linearly interpolated the values in the case of the read and write operations.

<table>
<thead>
<tr>
<th>Opinion</th>
<th>8 bits</th>
<th>16 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float add</td>
<td>0.2</td>
<td>0.4</td>
<td>0.9</td>
</tr>
<tr>
<td>float mul</td>
<td>0.6</td>
<td>1.1</td>
<td>3.7</td>
</tr>
<tr>
<td>R/W (&lt;8KB)</td>
<td>1.25</td>
<td>2.5</td>
<td>5.0</td>
</tr>
<tr>
<td>R/W (&lt;32KB)</td>
<td>2.5</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>R/W (&lt;1MB)</td>
<td>12.5</td>
<td>25.0</td>
<td>50.0</td>
</tr>
<tr>
<td>R/W (&gt;1MB)</td>
<td>250.0</td>
<td>500.0</td>
<td>1000.0</td>
</tr>
</tbody>
</table>

neural network into a resource constrained device. We are mostly interested in the resource consumption that will take place on the device. Nevertheless, as an additional side note we would like to mention that the algorithmic complexity of conversion into the CSR, CER and CSER representations is of $O(N)$, that is, of the order of number of elements in the matrix.

5.4.1 Experiments on simulated matrices

As first experiments we aimed to confirm the theoretical trends described in Section 5.3.

Efficiency on different regions of the entropy-sparsity plane

Firstly, we argued that each distribution has a particular entropy-sparsity value, and that the superiority of the different data structures is manifested in different regions on that plane. Concretely, we expected the dense representation to be increasingly more efficient in the upper-left corner, the CSR on the bottom-right (and along the right border) and the CER and CSER on the rest.

Figure 5.2 shows the result of performing one such experiment. In particular, we randomly selected a point-distribution on the $(H, p_0)$-plane and sampled 10 different matrices from that distribution. Subsequently, we converted each matrix into the respective dense, CSR, CER and CSER representation, and benchmarked the performance with regard to the 4 different measures described above. We then averaged the results over these 10 different matrices. Finally, we compared the performances with each other and respectively color-coded the max result. That is, blue corresponds to points where the dense representation was the most efficient, green to the CSR and red to either the CER or CSER. As one can see, the result closely matches the expected behavior.

Efficiency as a function of the column size

As second experiment, we study the asymptotic behavior of the data structures as we increase the column size of the matrices. From corollary 3.1 we expect that the CER and CSER formats increase their efficiency as the number of columns in the matrix grows (thus, as $n \to \infty$), until they converge to the same point, outperforming
Figure 5.2: The plots show the most efficient data structure at different points in the $H - p_0$ plane. The colors indicate the most efficient data structure at that point in the plane. We compare the dense data structure (blue), the CSR format (green) and the proposed CER/CSER data structures (red). We employed a $100 \times 100$ matrix and calculated the average complexity over 10 matrix samples at each point. The size of the set of unique elements was $|\Omega| = K = 2^7$. ©2020 IEEE.

the dense and sparse formats. Figure 5.3 confirms this trend experimentally with regard to all four benchmarks. Here we chose a particular point-distribution on the $(H, p_0)$-plane and fixed the number of rows. Concretely, we chose $H = 4.0$, $p_0 = 0.55$ and $m = 100$ (the later is the row dimension), and measured the average complexity of the data structures as we increased the number of columns $n \to \infty$.

As a side note, the sharp changes in the plots are due to the sharp discontinuities in the values of Table 5.1. For instance, the sharp drops in storage ratios come from the change of the index bit sizes, e.g., from $8 \to 16$ bits.

5.4.2 Compressed Neural Networks without Retraining

As second set of experiments, we tested the efficiency of the proposed data structures on quantized deep neural networks. In particular, we benchmarked their weight matrices relative to the matrix-vector operation, after them being quantized using two different methods: one where retraining of the network is required (Section 5.4.3) and one where it is not (Section 5.4.2). We treat them separately, since the statistics of the resulting quantized weight matrices are conditioned by the particular quantization method applied to them.

We start by first analyzing the latter case. This scenario is of particular interest since it applies to cases where one does not have access to the training data (e.g., federated learning scenario) or it is prohibited to retrain the model (e.g., limited access to computational resources).

In our experiments we firstly quantized the elements of the weight matrices of the networks in a lossy manner, while ensuring that we negligible impact their prediction accuracy. Similarly to (Choi, El-Khamy, and Lee, 2017; Choi, El-Khamy, and...
5.4. Experiments

Figure 5.3: Efficiency ratios of the different matrix representations compared to the dense data structure. $n$ denotes the column size. We chose a matrix with $H=4$, $p_0=0.55$ and fixed row size of 100. The results show the averaged values over 20 matrix samples. The size of the set of the elements was $|\Omega|=K=2^7$. The CER & CSER matrix formats tend to be more efficient as the column dimension of the matrix increases, and converge to the same value for $n \to \infty$ ©2020 IEEE.

Lee, 2020), we applied an uniform quantizer over the range of weight values at each layer and subsequently rounded the values to their nearest quantization point. That is, for each weight matrix $W$, we calculated the range of values $[w_{\min}, w_{\max}]$ (with $w_{\min}$ being the lowest weight element value and $w_{\max}$ analogously) and inserted $K = 2^b$ equidistant points inside that range, whose values were stored in the array $\Omega$. Then, we quantized each weight element in $W$ to its closest neighbor relative to $\Omega$ and measured the validation accuracy of the quantized network. In our experiments, we did not see any significant impact on the accuracy for all $b \geq 7$ (see Table 5.2). We chose the uniform quantizer because of it’s simplicity and high performance relative to other, more sophisticated quantizers such as entropy-constrained k-mean algorithms (Choi, El-Khamy, and Lee, 2017; Choi, El-Khamy, and Lee, 2020). Finally, we converted in a lossless manner the quantized weight matrices into the different data structures and tested their efficiency with regard to the four above mentioned benchmark criteria.

### Storage requirements

Table 5.2 shows the gains in storage requirements of different state-of-the-art neural networks. Gains can be attained when storing the networks in CER or CSER formats. In particular, we achieve more than x2.5 storage savings on the DenseNet architecture (Huang, Liu, and Weinberger, 2016), whereas in contrast the CSR data structure attains negligible gains. This is mainly attributed to the fact, that the dense and sparse representations store very inefficiently the weight element values of these networks. This is also reflected in Fig. 5.4, where one can see that most of the storage requirements for the dense and CSR representations is spent in storing the elements
Chapter 5. Compact and computationally efficient representations of DNNs

**Figure 5.4:** Storage requirements of a compressed DenseNet after converting its weight matrices into the different data structures. The weights of the network layers were compressed down to 7 bits (resulting accuracy is 77.09%, a drop of 0.11% from the original accuracy). The plots show the over the layers averaged result. Top: Compression ratio relative to the dense format. Bottom: Contribution of different parts of the data structures to the storage requirements ©2020 IEEE.

**Table 5.2:** Storage gains of models after their weight matrices have been compressed down to 7 bits and converted into the different data structures. The gains are relative to the original dense representation of the compressed weight matrices, and they show the over the layers aggregated results. The accuracy is measured on the validation set (in parenthesis is the accuracy of the original model) of the ImageNet classification task.

<table>
<thead>
<tr>
<th>Storage</th>
<th>Accuracy [%]</th>
<th>original [MB]</th>
<th>CSR</th>
<th>CER</th>
<th>CSER</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG16</td>
<td>71.63 (71.59)</td>
<td>553.43</td>
<td>x0.71</td>
<td>x2.11</td>
<td>x2.11</td>
</tr>
<tr>
<td>ResNet152</td>
<td>78.83 (78.31)</td>
<td>240.77</td>
<td>x0.76</td>
<td>x2.08</td>
<td>x2.10</td>
</tr>
<tr>
<td>DenseNet</td>
<td>77.02 (77.12)</td>
<td>114.72</td>
<td>x1.04</td>
<td>x2.74</td>
<td>x2.79</td>
</tr>
</tbody>
</table>

of the weight matrices $\Omega$. In contrast, most of the storage cost for the CER and CSER formats come from storing the column indices $colI$, which is much lower than the actual weight values.

**Number of operations**

Table 5.3 shows the savings attained with regard to number of elementary operations needed to perform a matrix-vector multiplication. As one can see, we can save up to 40% of the number of operations if we use the CER/CSER data structures on the DenseNet architecture. This is mainly due to the fact, that the dot product algorithm of the CER/CSER formats implicitly encode the distributive law of multiplications and consequently they require much less number of them. This is also
5.4. Experiments

**Figure 5.5:** Number of operations required to perform a dot product in the different formats for the experimental setup described in Fig. 5.4 (DenseNet). The CER/CSER formats require less operations than the other formats, because 1) they do not need to perform as many multiplications and 2) they do not need to load as many matrix weight elements ©2020 IEEE.

**Table 5.3:** Gains attained with regard to the number of operations, time and energy cost needed for performing a matrix-vector multiplication with the compressed weight matrices of different neural networks. The experiment setting and table structure is the same as in Table 5.2.

<table>
<thead>
<tr>
<th>#ops [G]</th>
<th>time [s]</th>
<th>energy [J]</th>
<th>original</th>
<th>CSR</th>
<th>CER</th>
<th>CSER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.08</td>
<td>x0.88</td>
<td>x1.40</td>
<td>x1.39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.37</td>
<td>x0.85</td>
<td>x1.27</td>
<td>x1.29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.70</td>
<td>x0.76</td>
<td>x2.37</td>
<td>x2.38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResNet152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.08</td>
<td>x0.93</td>
<td>x1.42</td>
<td>x1.41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.00</td>
<td>x0.93</td>
<td>x1.30</td>
<td>x1.31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.92</td>
<td>x1.25</td>
<td>x3.73</td>
<td>x3.74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DenseNet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.14</td>
<td>x1.11</td>
<td>x1.66</td>
<td>x1.65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.53</td>
<td>x1.10</td>
<td>x1.43</td>
<td>x1.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.51</td>
<td>x1.95</td>
<td>x6.40</td>
<td>x6.57</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

reflected in Fig. 5.5, where one can see that the CER/CSER dot product algorithms are mainly performing input load \((In_{load})\), column index load \((col_{load})\) and addition \((add)\) operations. Here, others refers to any other operation involved in the dot product, such as multiplications, weight loading, writing, etc. In contrast, the dense and CSR dot product algorithms require an additional equal number of weight element load \((\Omega_{load})\) and multiplication \((mul)\) operations.
Figure 5.6: Time cost of a dot product in the different formats for the experimental setup described in Fig. 5.4 (DenseNet). The CER/CSER formats save time, because 1) they do not require to perform as many multiplications and 2) they do not spend as much time loading the matrix weight elements ©2020 IEEE.

**Time cost**

In addition, Table 5.3 also shows that we attain speedups when performing the dot product in the new representation. Interestingly, Fig. 5.6 shows that most of the time is being consumed on IO’s operations (that is, on load operations). Consequently, the CER and CSER data structures attain speedups since they do not have to load as many weight elements. In addition, 20% and 16% of the time is spent in performing multiplications respectively in the dense and sparse representation. In contrast, this time cost is negligible for the CER and CSER representations.

**Energy cost**

Similarly, we see that most of the energy consumption is due to IOs operations (Fig. 5.7). Here the cost of loading an element may be up to 3 orders of magnitude higher than any other operations (see Table 5.1) and therefore, we obtain up to x6 energy savings when using the CER/CSER representations (see Table 5.3).

Finally, Table 5.4 and Fig. 5.8 further justify the observed gains. Namely, Table 5.4 shows that the effective number of shared elements per row of the network is small relative to the networks effective column dimension. To clarify, we calculated the effective number of shared elements by: 1) for all rows, calculate the number of shared weights, 2) aggregating the numbers and 3) dividing the result by the total number of rows that appear in the network. Similarly, the effective number of columns indicates the average number of columns in the network, and the effective sparsity level as well as effective entropy values indicate the over the total number of weights averaged result. Fig. 5.8 shows the distributions of the different layers of the networks on the entropy-sparsity plane where we see, that most of them lay in the regions where we expect the CER/CSER formats to be more efficient.
5.4. Experiments

Figure 5.7: Energy cost of a dot product in the different formats for the experimental setup described in Fig. 5.4 (DenseNet). Performing loading operations consumes up to 3 orders more energy than sum and mul operations (see Table 5.1). Since the CER/CSER formats need substantially less matrix weight element loading operations, they attain great energy saving compared to the dense and CSR formats ©2020 IEEE.

Table 5.4: Statistics of different neural network weight matrices taken over the entire network. \( p_0 \) denotes the sparsity level of the network, \( H \) the entropy, \( \bar{k} \) the number of shared elements per row, and \( n \) the column dimension. All neural networks have relatively low entropy, i.e., low number of shared elements compared to the column dimensionality.

<table>
<thead>
<tr>
<th></th>
<th>( p_0 )</th>
<th>( H )</th>
<th>( \bar{k} )</th>
<th>( n )</th>
<th>( \bar{k}/n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG16</td>
<td>0.07</td>
<td>4.8</td>
<td>55.80</td>
<td>10311.86</td>
<td>0.01</td>
</tr>
<tr>
<td>ResNet152</td>
<td>0.12</td>
<td>4.12</td>
<td>32.35</td>
<td>782.67</td>
<td>0.03</td>
</tr>
<tr>
<td>DenseNet</td>
<td>0.36</td>
<td>3.73</td>
<td>43.95</td>
<td>1326.93</td>
<td>0.03</td>
</tr>
<tr>
<td>AlexNet (Han, Mao, and Dally, 2016)</td>
<td>0.89</td>
<td>0.89</td>
<td>18.33</td>
<td>5767.85</td>
<td>0.01</td>
</tr>
</tbody>
</table>

5.4.3 Compressed Neural Networks with Retraining

In this section we benchmark the CER/CSER matrix representation on networks whose weight matrices have been compressed using quantization techniques where retraining was required in the process. This case is also of particular interest since highest compression gains can only be achieved if one applies such quantizations techniques on to the network (Han, Mao, and Dally, 2016; Wiedemann et al., 2019; Oktay et al., 2020). For instance, Deep Compression (Han, Mao, and Dally, 2016)
is a technique for compressing neural networks that is able to attain high compression rates without incurring significant loss of accuracy. It is able to do so by applying a three staged pipeline: 1) prune unimportant connections by employing algorithm (Han et al., 2015), 2) cluster the non-pruned weight values and refine the cluster centers to the loss surface and 3) employ an entropy coder for storing the final weights. Notice, that the first two stages aim to implicitly minimize the entropy of the weight matrices without incurring significant loss of accuracy, whereas the third stage lossless converts the weight matrices into low-bit representation. However, the proposed representation is based on the CSR format and, consequently, the complexity of the respective dot product algorithm remains on the same order. Concretely, the total number of operations that need to be performed is greater equal to the original CSR format. In contrast, the CER/CSER representations become increasingly efficient as the entropy of the network is reduced, even if the sparsity level is maintained (see Figs. 5.1 and 5.2). Hence, it is of high interest to benchmark their efficiency on highly compressed networks and compare them to their sparse (and dense) counterparts.

As first experiment we chose the AlexNet model (Krizhevsky, Sutskever, and Hinton, 2012), which was compressed using the Deep Compression technique. The overall entropy of the network was reduced down to 0.89 without incurring any loss of accuracy. Figure 5.9 shows the gains in efficiency when the network layers are converted into the different data structures. We see that the CER & CSER formats are able to surpass the dense and sparse representations at all four benchmark criteria. Therefore, CER/CSER data structures are much less redundant and efficient representations of highly compressed neural network models. Interestingly, the CER & CSER formats attain up to x14 storage and x20 energy savings, which is considerably higher than the sparse counterpart. Nevertheless, we do not attain significant speedups. This is due to the fact that, in our implementations, the time cost of loading the input elements was significantly higher than any other component.

\[\text{https://github.com/songhan/Deep-Compression-AlexNet}\]
5.4. Experiments

in the algorithm. This also explains why the CSR format shows similar speedups than the CER and CSER. However, this effect can be mitigated if one applies further optimizations on the input vector, such as data reuse techniques and/or better storage management of its values during the dot product procedure. With that we also expect significant gains in time performance relative to the CSR format, as reflected by the energy measures.

Lastly, we trained and compressed additional architectures while following a similar compression pipeline as described in (Han, Mao, and Dally, 2016). Concretely, we: 1) pretrained the architectures until we reached state-of-the-art accuracies, 2) sparsified the architectures using the technique proposed in (Molchanov, Ashukha, and Vetrov, 2017), 3) applied an uniform quantizer to the non-zero values in order to reduce their effective bit size, and 4) converted the weight matrices into the different representations and benchmarked their efficiency relative to their matrix-vector product operation. We chose to benchmark the same architectures as reported in (Molchanov, Ashukha, and Vetrov, 2017; Louizos, Ullrich, and Welling, 2017). That is, an adapted version of the VGG network\(^6\) for the CIFAR-10 image classification task and the fully connected and convolutional LeNet architectures for the MNIST classification task. The respective accuracies and compression gains can be seen in Tables 5.5 and the gains relative to the dot product complexity in Table 5.6. As we can see, we attain significantly higher gains in all four benchmarks when we convert their weight matrices into the CER & CSER representations. In particular, we are able to attain up to x42 compression gains, x5 speedups and x90 energy gains on the VGG model.

\(^6\)http://torch.ch/blog/2015/07/30/cifar.html.
Table 5.5: Storage gains of different models after they have been compressed as described in Section 5.4.3. The VGG model was trained on CIFAR-10 as benchmarked in (Molchanov, Ashukha, and Vetrov, 2017; Louizos, Ullrich, and Welling, 2017). The LeNet architectures were trained on MNIST as benchmarked in (Molchanov, Ashukha, and Vetrov, 2017; Louizos, Ullrich, and Welling, 2017). The accuracy column (Acc) shows the accuracies of the compressed (parenthesis) models. Finally, the sparsity column (sp) displays the ratio between the non-zero weight values and the total number of weight elements.

<table>
<thead>
<tr>
<th>Storage</th>
<th>Acc [%]</th>
<th>sp [%]</th>
<th>orgnl [MB]</th>
<th>CSR</th>
<th>CER</th>
<th>CSER</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-CIFAR10</td>
<td>90.13 (91.54)</td>
<td>4.28</td>
<td>59.91</td>
<td>x17.00</td>
<td>x41.95</td>
<td>x41.59</td>
</tr>
<tr>
<td>LeNet-300-100</td>
<td>97.16 (98.32)</td>
<td>9.05</td>
<td>1.06</td>
<td>x8.00</td>
<td>x19.52</td>
<td>x18.98</td>
</tr>
<tr>
<td>LeNet5</td>
<td>98.27 (99.44)</td>
<td>1.90</td>
<td>1.722</td>
<td>x35.08</td>
<td>x73.16</td>
<td>x72.62</td>
</tr>
</tbody>
</table>

Table 5.6: Gains attained (aggregated over layers) with regard to the number of operations, time and energy cost needed when benchmarking the matrix-vector multiplication of the weight matrices of the networks described in Table 5.5. The performance gains are relative to the original dense representation of the compressed weight matrices.

<table>
<thead>
<tr>
<th>#ops [M]</th>
<th>time [ms]</th>
<th>energy [mJ]</th>
<th>original</th>
<th>CSR</th>
<th>CER</th>
<th>CSER</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-CIFAR10</td>
<td>878.38</td>
<td>208.00</td>
<td>139.64</td>
<td>x3.71</td>
<td>x5.53</td>
<td>x5.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x3.63</td>
<td>x5.09</td>
<td>x5.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x35.41</td>
<td>x89.81</td>
<td>x90.34</td>
</tr>
<tr>
<td>LeNet-300-100</td>
<td>1.065</td>
<td>0.25</td>
<td>0.02</td>
<td>x9.54</td>
<td>x12.73</td>
<td>x12.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x9.76</td>
<td>x11.61</td>
<td>x11.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x14.23</td>
<td>x54.46</td>
<td>x54.10</td>
</tr>
<tr>
<td>LeNet5</td>
<td>7.59</td>
<td>1.94</td>
<td>0.48</td>
<td>x3.61</td>
<td>x4.15</td>
<td>x4.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x3.52</td>
<td>x3.54</td>
<td>x3.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x60.90</td>
<td>x87.49</td>
<td>x96.58</td>
</tr>
</tbody>
</table>
5.5 Summary

In this chapter we presented two novel matrix representations, Compressed Entropy Row (CER) and Compressed Shared Elements Row (CSER), that exploit the fact that the matrix has low first-order entropy statistics in order to reduce both, its storage requirements as well as the algorithmic complexity for performing the dot product operation. We showed on an extensive set of experiments that the CER & CSER formats are more compact and computationally efficient as compared to traditional matrix representations, such as dense or sparse CSR. In particular, we attained up to $\times 42$ compression ratios and $\times 13$ reduction of computational complexity after converting the weight matrices of sparse and quantized DNN models to the CER & CSER formats.

5.6 Limitations

Although the CER & CSER representations reduce the memory and computational complexity of the inference algorithm at a fundamental level, dedicated compilers and/or hardware platforms need to be implemented in order to be able to attain efficiency gains in practice. Factors such as parallelism, data locality, control flow and data movement need to be taken into account, which greatly impact the resource consumption of the algorithm.

Nevertheless, the in this chapter introduced metric of complexity closely reflects the costs spend when the data is processed in a sequential manner. Hence, we expect that we will be able to fully exploit the CER and CSER representations on practical level if specialized hardware akin to (Han et al., 2016; Chen et al., 2019; Parashar et al., 2017) is designed. That is, an accelerator that implements an efficient data flow of the CER and CSER representations, combined with processor units where data is inherently processed in a sequential manner and tailored to execute CER & CSER data formats.
Chapter 6

Software-Hardware co-design: Specialized hardware accelerator for compact deep neural networks

In the previous chapter 5 we discussed how we can take advantage of the low entropy statistics of the DNN parameters in order to reduce their memory requirements as well as their computational complexity for performing inference. However, the introduced metric served only as an approximation, informing only of the potential efficiency gains that can be attained by executing compact DNN models. Eventually, latency and power consumption of inference must be measured on a physical hardware platform in order to benchmark the efficiency of a particular model.

The problem is that these measurements are highly conditioned on the particular hardware platform the models are executed on, and on the manner the models have been compiled. Since most common hardware platforms such as CPUs and GPUs are designed for generic purposes, they cannot entirely exploit the unique properties that emerge from compact models, as such limiting the potential efficiency gains that can be attained by running such models.

Therefore, the topic of this chapter focuses around the question of, how we can design hardware architectures that are able to efficiently exploit and execute compact forms of DNN models. Concretely, we propose a hardware architecture that is able to leverage the particular properties that emerge from parameter sets with low first-order entropy statistics: (1) sparsity, (2) low cardinality of the set of unique values and (3) a high number of repeated and consecutive values. We exploit all these factors in order to attain very high area efficiency for performing inference.

6.1 Introduction

As mentioned in chapter 1, there is a growing demand for deploying DL models to edge devices where the data is measured and/or collected. Running models directly on the edge has several advantages in terms of latency, energy, costs and privacy issues, the latter becoming increasingly important due to stricter data privacy regulations being applied in many countries (Voigt and Bussche, 2017). However, edge devices have usually tight resource constraints, challenging the deployment of state-of-the-art DNN models.

In the previous chapters 3, 4 & 5 we discussed techniques that are able to optimize the complexity of DNN models by devising compressed representations. However, as briefly discussed in chapter 2, running compact representations of data on...
Chapter 6. Software-Hardware co-design: Specialized hardware accelerator for compact deep neural networks

general-purpose hardware platforms in an efficient manner comes with a series of challenges that can not always be overcomed. Firstly, the compressed representations introduced in chapters 4 & 5 have, by design, inherent dependencies between the data elements, which greatly reduce the parallelization capabilities. In particular, the CER & CSER representations (introduced in chapter 5) require a level of indirection with regards to accessing the data in memory, which renders poor locality and can not be exploited efficiently on common hardware platforms such as CPUs, GPUs and/or MCUs. In addition, data that has been compressed by an entropy coder as in DeepCABAC (Wiedemann et al., 2020a) (chapter 4), poses control flow and data movement challenges since the data is not byte-aligned, not to mention that sequential decoding is inherently required due to the context-based adaptation property of the entropy coder. This motivates the study of novel hardware architectures that are able to (at least partially) overcome some of the mentioned challenges and support the efficient execution of compact representations of DNNs.

In this chapter, we propose one such specialized hardware architecture. We named it FantasIC4, since it requires parameter sets of 4bit precision or less. We apply several optimization techniques, all tailored to mainly increase the area efficiency (GOPS/mm²) and lower the power consumption for performing inference. In particular, FantasIC4 is designed for the efficient execution of compact representations of multilayered perceptrons (MLPs). We focus on MLPs since they are the status quo models for use-cases with very tight resource constraints, such as in the IoT. As stated by (Wang et al., 2020), MLPs have been successfully used in a wide range of application scenarios, such as disease detection, activity recognition, and brain-machine interface. Many studies identified MLPs to be the best or one of the best algorithms to solve tasks in the IoT domain using wearable devices. This fact stresses the importance of having high area efficiency and low power consumption for inference, since both are a tight resource in those use-cases. Also, reducing area requirements does automatically reduce the production costs, which is desirable if such an accelerator ought to be deployed to, e.g., millions of IoT devices. Although here we focus on MLPs, we want to stress that our hardware design can be extended to support a wider set of popular DNN architectures that are composed by fully-connected layers, such as LSTMs and Transformers.

In the following section 6.3, we will describe in more detail the rationale behind our particular optimization paradigm, whereas in section 6.4 we provide an in-depth description of FantasIC4s hardware architecture. In section 6.6 we show results from benchmarking FantasIC4 efficiency on several models, which show that we can attain up to ×51 gains in terms of throughput and ×145 in terms of area efficiency on particular ML tasks as compared to previous work.

6.2 Related work

There are large number of hardware accelerators from both, the academia and industry, that are focus on achieving high performance as well as energy efficiency. Some of the topics that have been studied and analyzed are:

Data Flow Movement. Data flow movement is one of the key aspects in designing the hardware accelerators for any AI applications. Effective movement of the weights and activations help in reducing a large amount of energy and the power requirement. (Chen et al., 2019) provides an effective row stationary method and competent reusing of weights, input feature maps (Ifmaps) and partial sums (Psums). (Shivapakash et al., 2020) was able to increase execution efficiency by truncating
6.3 Rationale behind FantastIC4’s design

Psurns from each of the preceding layer and performing inference on the truncated Psurns. Bit Fusion (Sharma et al., 2018) dynamically varied the weights across the different layers of a DNN models.

In contrast, FantastIC4 concentrates on reducing the data movement by considering only 4bit precision of the parameter values and handling FIFOs effectively. The later is achieved thanks to the proper handling of sparse data. Moreover, floating point operations are pipelined to ensure that dynamic power is saved without compromising on the accuracy.

**Systolic Arrays and Bit Serial Computations.** Systolic arrays were first reported in (Jouppi and et.al, 2017) for the efficient computation of the DNN models and to reduce the unwanted power consumption in performing the MAC and matrix vector multiplications. The need for expensive multiplications was reduced by using binary weights in (Andri et al., 2016). The bit serial MAC operation was used in (Lee et al., 2018) to conserve the energy in the MAC operation and save the core area of the chip.

FantastIC4 increases computation efficiency by reducing the overall number of multiplication down to only four per output element. This is accomplished by replacing the traditional MAC operation by an “accumulate-and-multiply” computational paradigm instead.

**Sparse Data Compression.** (Han et al., 2016) exploits on compressed representations of DNNs that were sparsified and quantized. It proved gains of $\times 19$ in energy efficiency when compared to uncompressed versions. The compression was further extended to convolutional layers in (Parashar et al., 2017; Chen et al., 2019). In all three accelerators, CSC formats were used in order to represent sparse data. The scalpel accelerator (Yu et al., 2017) showed that weight pruning achieves a total speedup of $\times 1.9$.

In contrast to FantastIC4, all mentioned accelerators support only one particular compressed format which can greatly limit the attainable compression gains and consequently the latency and power improvements from off- to on-chip data movement.

**FPGA based Accelerators.** A plethora of FPGA accelerators have been proposed in the literature for optimizing the DNNs inference-efficiency. (Duan et al., 2019) build a specialized inference engine for CNNs with binary weights, attaining high energy-efficiency. The processor achieves a throughput of 2100 GOPs with a latency of 4.6ms and power of 28W. The hardware-software co-design library to efficiently accelerate entire CNNs and MPLs on FPGAs was shown in (Zhang et al., 2019). The floating point arithmetic CNN accelerator (Lian et al., 2019) introduced an optimized quantization scheme based on rounding and shifting-operations, they reported the overall throughput of 760.83 GOPs. Other accelerators worked on sparse matrix vector multiplications mainly for MLPs (Wang et al., 2018; Shi et al., 2019). Even though these accelerators have a good performance, they still lack either in throughput, power or latency requirement. The FantastIC4’s FPGA version utilizes efficient computation approach to achieve high throughput, with minimal power, latency and resource requirements.

6.3 Rationale behind FantastIC4’s design

In this work, we propose to apply several optimization techniques that, in combination, are tailored to reduce both, area and energy requirements for performing
Chapter 6. Software-Hardware co-design: Specialized hardware accelerator for compact deep neural networks

Figure 6.1: Sketch example on the different computational paradigms when performing the dot product algorithm. Given two input vectors, the multiply-accumulate (MAC) calculates the respective scalar product by firstly multiplying the elements and subsequently adding them. In contrast, the accumulate-multiply (ACM) firstly sums the elements of one of the vectors (in this diagram the right-hand-side vector) according to the bit-decomposition of the other, then multiplies the respective basis values and, finally, reduces the output. In the above sketch the base values were \([-1.43, -0.77, 0.13, 2.53]\), and we color-coded them according to \([\text{blue, green, red, pink}]\) respectively. Each binary array resulting from the bit-decomposition corresponds to the coefficient of each vector element, respective to the base value. Thus, the original element values result by performing the linear combination in the vertical direction, for instance, \(-2.2 = 1 \times (-1.43) + 1 \times (-0.77) + 0 \times (0.13) + 0 \times (2.53)\) ©2021 IEEE.

Inference. The main idea is to apply techniques that minimize the memory requirements as well as the number of multiplications needed to perform inference, since both are the major source of area utilization and power consumption.

6.3.1 4bit quantization of the parameter set

As thoroughly discussed in previous chapters 3, 4 and 5; quantization is a powerful technique for lowering the memory as well as the computational resources for inference. The increasing demand for deployment of DNNs on edge devices with very tight hardware constraints (e.g. microcontrollers) has pushed researchers to investigate methods for extreme quantization, resulting in weights with merely 4bits or lower (Sze et al., 2020; Deng et al., 2020). This directly translates to \(\times 8\) compression of the model, which is beneficial for minimizing the costs involved in off- and on-chip data movement of the weights. In particular, FC layers have shown to be highly redundant and robust to extreme quantizations down to 4bit (Han, Mao, and Dally, 2016; Han et al., 2016), which motivated the particular design of our work.

Contribution 1) Increasing the computational efficiency

However, most often the inference modules of extremely quantized DNNs are implemented following the usual multiply-accumulate (MAC) computational paradigm (see figure 6.1). We argue that in the regime of extreme bit-width reduction this computational paradigm is not the most efficient. Instead, we propose to first accumulate the activations at each bit-level and subsequently multiply the results, thus
6.3. Rationale behind FantastIC4’s design

an accumulate-multiply (ACM) computational paradigm (see figure 6.1), akin to the computational flow of the CER & CSER formats introduced in chapter 5. More concretely, we follow the equation

$$W \cdot A = \left( \sum_{i=0}^{3} \omega_i B_i \right) \cdot A = \sum_{i=0}^{3} \omega_i (B_i \cdot A)$$ (6.1)

where we denote as $W$ the weight parameters of, e.g., a fully-connected layer, $A$ the input activations, $\cdot$ the operator denoting the dot product and $B_i$ a binary mask corresponding to the base $\omega_i$. Thus, as shown in equation (6.1), we represent the weight parameters $W$ as a linear combination of four binary masks $B_i$ with respective coefficients $\omega_i$. This representation generalizes any type of 4bit-representation that is applied to the weights. For instance, if $\omega_i = 2^i$ then the elements of $W$ are simply represent in uint4 format.

As discussed in chapter 5, this computational paradigm can significantly reduce the required number of multiplications. To recall, the cost of performing multiplications grows approximately quadratically with the number of bits as opposed additions, which grow linearly, corroborated by the energy values in table 5.1 (Horowitz, 2014). Concretely, let $(m \times k)$ and $(k \times n)$ be the dimensions of $W$ and $A$ respectively, then the cost of multiplications for an ACM approach is $O(4mn)$ vs $O(mkn)$ being the cost if we follow the MAC computational paradigm. If the dimension along the scalar product $k$ is large enough, then it pays off to follow the ACM computational flow.

(Contribution 2) Increasing the capacity of the model

Moreover, the usual MAC computational paradigm requires to also quantize the activations of the model down to 4bits in order to exploit the benefits from extreme quantization. Since activations are more sensitive to perturbations than the weights (see figure D.1 in appendix D), this most often results in significant degradation of the NN prediction performance. Moreover, special parameters such as bias and batch-normalization tend to also be more sensitive than the weight parameters. This motivates the support of mixed-precision layers where input and output activations, as well as bias and batch-norm parameters can be represented with higher precision than the weights in order to compensate for the accuracy degradation. FantastIC4s design supports higher precision activation values, since this can be easily integrated within the ACM computational flow. In addition, we support full-precision representation of the batch-norm parameters as well as the bias coefficients, since their memory and compute cost are relatively low as compared to the operations involved in the weight parameters, in particular for FC layers. Lastly, we also do not constrain the linear coefficients values $\omega_i$ to be of powers of 2, as it is most common in the MAC approach, but allow $\omega_i \in \mathbb{R}$. This increases the expressive power of $W$, and with it the capacity of the model, allowing it to better learn more complex tasks.

6.3.2 Low-entropy statistics of the parameter set

Now we discuss how FantastIC4 takes advantage of low-entropy statistics in order to reduce memory as well as computational cost.
Chapter 6. Software-Hardware co-design: Specialized hardware accelerator for compact deep neural networks

Table 6.1: Control States of the FantastIC4 Control Unit

<table>
<thead>
<tr>
<th>Data Movement</th>
<th>-</th>
<th>Acts,Wt,Bias, alpha and CSR</th>
<th>CSR Data</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>-</td>
<td>-</td>
<td>BM Conv</td>
<td>Wt ID</td>
<td>Add tree/MAC</td>
<td>Fix-Flt</td>
<td>FLT</td>
<td>Mul1</td>
<td>Mul1</td>
</tr>
<tr>
<td>State</td>
<td>Start</td>
<td>State1</td>
<td>State2</td>
<td>State3</td>
<td>State4</td>
<td>State5</td>
<td>State6</td>
<td>State7</td>
<td>State8</td>
</tr>
<tr>
<td>Time(ns)</td>
<td>0</td>
<td>5000</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>50</td>
<td>50</td>
<td>40</td>
</tr>
</tbody>
</table>

(Contribution 3) Saving arithmetic operations and dynamic energy consumption

As thoroughly explained in chapter 2, low-entropy induces sparsity which can be exploited in order to save computations. In particular, FantastIC4 does not perform additions of activations when a zero-valued weights are present, thus saving on arithmetic operations and consequently dynamic energy consumption.

Moreover, low-entropy statistics induce frequent and consecutive appearance of the same element value. FantastIC4 leverages this fact by saving switching-operations when loading equal non-zero element values, thus consequently saving on dynamic power consumption.

(Contribution 4) Multiple lossless compression

Naturally, by reducing the entropy of the weight distribution we automatically make the models more compressible. In this work we take advantage of sparse representation, since they are easier to handle from a hardware architecture perspective. However, we consider the integration of more sophisticated compressed formats as introduced in chapters 4 and 5 in future work.

In chapter 5 we already introduced the Compressed Sparse Row (CSR) format as an efficient manner to compress sparse parameters. To recall, the CSR format is based on applying run-length coding for saving the signaling of the positions of non-zero values. However, another manner to compress sparse parameters is by applying a simple form of Huffman coding (Huffman, 1952), which consists of storing a bit-mask indicating the positions of the non-zero values, followed by an array of non-zero values organized in, e.g., row-major order. In the high sparsity regime (>90% of zeros), the CSR format attains higher compression gains than the Huffman code, whereas for smaller sparsity ratios (25% - 90% of zeros) the Huffman code compresses more the weights. Since the sparsity ratio of different layers can vary significantly, FantastIC4 supports the on-chip processing of both sparse representations, thus not requiring a priori decoding. This allows for more flexible compression opportunities, consequently boosting the compression gains of the model and saving on off-to-on-chip transmission costs.

In summary, FantastIC4 leverages the fact that the weight parameters of DNNs can be quantized down to 4bits and have low-entropy statistics, in order to drastically reduce the area requirement and power consumption by means of compression and reduction in number of multipliers.

In the following section 6.4 we will discuss in more detail how FantastIC4 hardware design integrates all mentioned properties.

6.4 FantastIC4: Efficient hardware accelerator for 4bit MLPs

Figure 6.2 shows the overview of the FantastIC4 system. The whole system is a heterogeneous combination of CPU and an FPGA architecture. The entire system comprises of mainly three parts: the software program on the CPU, the external DDR3
6.4. FantastIC4: Efficient hardware accelerator for 4bit MLPs

The software part mainly consists of the CPU that transfers the input data as well as the DNN model (only one time) to the FPGA. Since all the data is usually very large and can therefore not entirely be stored on an on-chip BRAM, some of it is stored in an off-chip DRAM. The data is then accessed through a memory controller which is built across a memory interface generator (MIG) IP. On the FPGA chip, we have the FantastIC4 control unit, memory controller, I/O Buffers and the FantastIC4 accelerator. The memory controller facilitates the movement of the input data from off-chip DRAM to the accelerator and stores back the computation results into the DRAM. The control unit regulates the behaviour of other modules on the FPGA, it handles the data movement and the computation inside the accelerator. The I/O buffers stores the input data for processing and stores back the PSum data from the accelerator for the next layer inference. FantastIC4 accelerator is the heart of the entire system which reads the data from the DRAM, perform the computation and stores back the results into the DRAM memory.

6.4.1 Memory Controller and Input/Output Buffers

The DDR3 memory is accessed by the FantastIC4 accelerator through a MIG interface operating at a clock frequency of 200MHz. We employ the AXI communication protocol for the data movement between the FPGA chip and the off-chip DRAM.
Chapter 6. Software-Hardware co-design: Specialized hardware accelerator for compact deep neural networks

The microblaze CPU and other AXI control IPs are used to communicate through the MIG interface with the DDR3. The memory controller receives the instruction from the FantastIC4 control unit through the AXI master to read and write the data from/to the memory. The I/O buffers provides the dual buffering for the data movement in a ping-pong manner.

6.4.2 FantastIC4 Control unit

Our proposed accelerator has two levels of control hierarchy. Table 6.1 shows the control states for our accelerator. The first level of hierarchy i.e. the start and the state1 controls the data movement between the DRAM, memory controller and the accelerator on the FPGA chip. Here the activations, weights, bias, alpha values for floating point operation, FIFO data and 256bits CSR Pointer data are moved into their respective memory/registers for computation. In this level all the data movement operations are performed sequentially, the total time taken to complete these two states are approximately around 5000ns. Here the total time taken is mainly dependent on the DNN model which is under inference. The next level of hierarchy we perform the computations, state2-state9 shows the different stages of processing performed on the accelerator. The different orders of computation performed are: CSR to bit-mask conversion, weight ID generation, accumulation and multiply operation and finally the single precision floating point operation. The total time taken to perform the computation is around 220ns. The computation time is less because all the states are working concurrently and each state is independent on the other states except on the first iteration.

6.4.3 FantastIC4 Architecture

The top-level hierarchy of the FantastIC4 architecture is shown in Fig. 6.3. The architecture operates on a single clock frequency domain of 150MHz (FPGA Based Implementation) and 800MHz (ASIC Based Implementation). FantastIC4 is composed of CSR to bit mask logic to perform CSR to bit mask conversion, FIFO modules to store the weight IDs for 256 adder trees, weight ID generator fetches the data from the FIFO modules based on the outcome of CSR to bit mask conversion. Adder tree performs the accumulation of the activations based on the weights IDs from the ID generator. The MAC array performs 4 multiplication and 3 addition operations. Fixed point to floating point converter converts a 16bit fixed point MAC output into a 32bit single precision floating output. 32bit floating point MAC output will be multiplied by a 32bit alpha1 values, the output of the multiplier1 will be added with the bias. The output of the adder will undergo a non-linear activation operation called ReLU, to perform the computation \( f(x) = \max(0, x) \). Final floating point multiplication is performed with a 32bit alpha2 value and then the 32bit floating point value will be rounded back to 16bit integer value to generate the final PSum.

**CSR to bit-mask Logic.** By default, FantastIC4 loads the positions of the non-zero elements of a row of the sparse weight matrix according to the compressed Huffman representation, which consists of a simple binary mask of width 256. The bit-mask controls the weight ID movement into the adder tree. However, when a layers non-zero positions are compressed following the CSR format, a logic must be implemented that converts them back to a bit-mask representation, which is the purpose of the CSR to bit-mask Logic. The conversion logic is shown in Fig. 6.4, the compressed non-zero position data pointers comprising of 256 bits will be splitted into chunks of 32 which is of 8bit wide. Based on the 8bit value, each bit of the
encoded bit-mask will be set to ‘1’. For ex: As shown in Fig. 6.4, the 0th chunk had a value of 241 and
31st chunk had a value of 51. So the corresponding 241st bit and 51st bit will be set to 1 and the remaining bits will be set to 0 to generate a 256bits encoded bit-mask data. Both the CSR pointer data and bit-mask data will be selected from a ×2 1 Mux through a “Select Bits” to generate the final encoded data for weight ID generation.

**FIFO Module and Weight ID Generator.** The FIFO module has 256 individual FIFOs which has a width of 4 and depth of 256, each storing the non-zero weight elements of a particular column of the weight matrix of the layer. These FIFO modules are stored in array of registers. The weight ID generator has a simple selection logic, where each individual IDs of 4bits are fetched from the FIFOs based on the encoded bit-mask data. If the encoded bit-mask is ‘1’, then an ID will be fetched from the FIFOs or else the pointer points to the same location of the fetched data. The weight ID generator has a cluster of 256 ID modules, which store the 4bit IDs from the FIFO if the 256 individual bits from the bit-mask is ‘1’ or else it stores a 4bit zero data.

**Adder Tree and MAC Array.** An adder tree comprises of an array of 256 adders arranged in a logarithmic fashion. The adder tree is grouped into two stages: Adder Stage1 and Adder Stage2. Adder Stage1 has 128 adders arranged in a single group. Each adder in the stage1 has three levels of hierarchy with a control parameter in each hierarchy. The Fig. 6.5 shows the adder schematic in the Stage1. The adders are fed with the two-different activations and two-different IDs from the weight ID generator. All the activations in the adder tree are static and it is used for all cycles of computations. The static activations in the adder tree saves significant power consumption. By having a static activations inside the adder rather than accessing it from the memory saves up to 15% of power consumption. In the level1 hierarchy, the 4bit weight IDs control the movement of the activations inside the adder. Each bit from the weight IDs forms a channel, that regulate the flow of activation to the level2. If the ID is 1, then the 16bit activation is fed or else a zero value is fed to level2. There will be a total of eight groups of activation data coming out of the level1 hierarchy. In level2 hierarchy, the activation switches between the upper and lower half of 8bits. This technique is employed to fit the larger networks into the hardware and improvise the prediction in the hardware. In this hierarchy, if the activation switch is low, a lower half of the bits are selected or else the upper half is selected. In the level3 hierarchy, the actual computation is performed. The sign mode determines, whether the activations need to be added or subtracted. Finally the four different computations are performed among the eight groups, to generate
the four output data of 16 bits from each adder. The Adder Stage2 has 128 adders arranged in a multiple group. The first group in adder stage2 has 64 adders, second group has 32 adders and similarly other groups are scaled down logarithmically. The adders in adder stage2 performs only the computation, unlike the adders in stage1. Based on the sign-bit in the output data from the stage1, either addition or subtraction is performed.

The MAC array performs four multiplications and three additions respectively. The four outputs each of 16 bits from the adder tree will be multiplied with the 16 bit basis weights to generate a 32 bit product, which we will be accumulated to generate the final 32 bit MAC output.

**Floating Point Operations.** The floating point operation mainly comprises of fixed to floating point conversion, floating point multiplications, floating point addition and final 32 bit floating point to 16 bit integer conversion. In the fixed to floating point conversion, a 32 bit fixed point MAC data is converted into equivalent single precision floating point data as shown in Algorithm 3, a leading one will be detected from the MAC output and corresponding conversion operation is performed.

The converted floating point data will undergo a single precision floating point multiplication with Alpha1 values. The Alpha1 values are stored in a SRAM of 1KB. With this scaling factors FantastiC4 is able to accommodate for de-quantization as well as batch norm parameters. As shown in Fig. 6.6, both the inputs will be normalized and split into its equivalent sign, mantissa and exponent part. The 23 bits mantissa will be multiplied with each other to generate 48 bit output, the MSB of the multiplied output will be used to calculate the final mantissa and the exponent part. The sign bits of both the inputs will be XORed to generate the final sign bit.

The final sign, exponent and mantissa will be concatenated to generate the final 32 bits multiplied output. Subsequently, the multiplied floating point will be added with the bias data stored in another 1KB SRAM. This operation is similar to the multiplication operation in terms of normalization of the data. Then, the added data will undergo an nonlinear activation ReLU function as \( f(x) = \max(0, x) \), since it is the status quo non-linear function for most MLP models. The ReLUed output will be further multiplied with a single 32 bit Alpha2 value to generate the final 32 bit multiplied output. These scaling factors take further quantization parameters into account, important for the correct calibration of the subsequent quantization step, which consists of a final rounding of 32 bits to a 16 bit integer. The 16 bit integer is the
Algorithm 3 Fixed Point to Floating Point Conversion

mac\_out: 32bit fixed point MAC output
convert\_out: Single precision floating point number

procedure FIXEDTOFLOAT(mac\_out, convert\_out)

lod ← Leading one detector
convert\_out\_sign ← Sign bit of floating point
convert\_out\_exponent ← Exponent of floating point
convert\_out\_mantissa ← Mantissa of floating point

for k ← 30 downto 0 do
    if (mac\_out[k] == 1) then
        lod = k;
        convert\_out\_sign = mac\_out[31];
        convert\_out\_exponent = lod + 127;
        convert\_out\_mantissa = mac\_out << (23 - lod);
    Combine sign, exponent and mantissa to generate convert\_out

return convert\_out

---

**Figure 6.6: Floating Point Multiplier ©2021 IEEE.**

---

6.5 Training 4bit-compact DNNs

As described in the previous sections 6.3, our proposed optimization paradigm is based on the fact that the weight parameters exhibit low-entropy statistics and can be represented with 4bits. However, if we naively lower the entropy and strongly quantize a pretrained model then, most often than not, we would incur a significant drop in accuracy. Therefore, in this work we apply the entropy-constrained training method described in chapter 3.3 in order to minimize their entropy while simultaneously making them robust to extreme quantization.
6.6 Experiments

6.6.1 Experimental setup

Datasets & Models

In the experiments section we distinguish between hardware-conform and non-conform models. Conform models are those that are fully compatible with our hardware architecture, thus the entire end-to-end inference procedure can be performed on it. Consequently, conform models include only FC layers with up to 512 input/output features. Optionally, BatchNorm layers are allowed which can result in accuracy gains.

To cover a variety of use-cases with the conform models, we trained and deployed several models solving classification tasks for audio, image, biomedical and sensor data. Concretely, we considered the task of hand gesture recognition (HGR) based on the biomedical and sensor dataset described in (Georgi, Amma, and Schultz, 2015), the google speech commands (GSC) (Warden, 2018) dataset for the task of audio classification, and MNIST (LeCun and Cortes, 2010) and CIFAR-10 (Krizhevsky, 2009) datasets for small-scale image classification task. We trained and implemented custom and well-known MLPs for solving the above mentioned tasks. In addition, in order to benchmark our quantization algorithm, we also used non-conform models, which we have not trained ourselves but obtained from publicly available sources. Concretely, ResNet-50 and -34 come from the torchvision model zoo 1, EfficientNet-B0 from 2 and ResNet-20 from 3. We trained further these models by applying the entropy-constrained method described in 3.3, and benchmarked their accuracies at different regularization strengths. We refer to appendix D for a more detailed description of the datasets, and the training and model architecture of our custom MLPs.

Hardware simulation setup

The proposed FantsatIC4 accelerator was synthesized using Synopsys Design Compiler (DC) under the GF 22nm FDSOI SLVT technology. We placed and routed the design using Synopsys IC compiler (ICC2). After the sign-off and RC extraction using STARRC, we performed the timing closure using Synopsys Prime-Time. We annotated the toggle rates from the gate level simulation and dumped the toggling information into Value Change Dump (VCD) file and estimated the power using Prime-Time.

6.6.2 Benchmarking training of 4bit-compact DNNs

In chapter 3.3 we showed results from applying the entropy-constrained training method, configured in a manner so that ternary & sparse networks result from the algorithm. Here we test if increasing the capacity of the model to 4bits (thus, 16 clusters instead of only 3) indeed leads to better Pareto-optimal results. Moreover, thanks to the support of full-precision scaling factors which can accommodate for batchnorm parameters we expect our DNN models to be more robust to strong

---

1https://pytorch.org/docs/stable/torchvision/models.html
2https://github.com/lukemelas/EfficientNet-PyTorch
3https://github.com/akamaster/pytorch_resnet_cifar10, Yerlan Idelbayev’s ResNet implementation for CIFAR10/CIFAR100 in PyTorch
quantization & sparsification. Figure 6.7 shows this phenomena. We can see that our 4bit DNN models reach better Pareto-optimal fronts with regards to accuracy vs sparsity, as compared to our previous work EC2T presented in (Marban et al., 2020).

To further benchmark our training method against previous work, in table 6.2 we show the prediction performance of our models on the datasets described above and summarize the results attained by other authors. We can see that we consistently attain similar or higher prediction performances. Table 6.2 also shows the benefits of applying a hybrid compression scheme as opposed to the single compression format approach recommended in previous work. To recall, our compression scheme encodes each layer by applying the CSR, the simple Huffman code (or bitmask format), and the trivial 4bit dense representation, and chooses the most compact representation between them. We see, that we attain about x2.36 boost in compression gains on average as compared to the CSR-only approach proposed by (Han et al., 2016; Chen et al., 2019), and ×1.77 higher compression rates than the trivial 4bit dense format. These gains directly translate to reduction in memory and therefore area requirements, as well as in reduction of energy costs spend in off- to on-chip data movement, which stresses the importance of supporting multiple representations.

6.6.3 Benchmarking hardware efficiency

Latency and throughput calculation

In the literature, latency and/or throughput are usually reported as the number of floating point operations per second (OPS) an accelerator can process (Sze et al., 2020). Since in this work we focus on MLPs, we can approximate the throughput calculation by considering the number of clock cycles involved in performing a matrix-vector multiplication (MVM).
Table 6.2: Comparison of the FantastIC4-quantization approach vs state-of-the-art 4-bit quantization techniques. Unless otherwise specified, all approaches quantize all network layers, including input- and output-layers, excluding batch normalization- and bias-parameters.

<table>
<thead>
<tr>
<th>Model</th>
<th>Org. Acc. (%)</th>
<th>Acc.</th>
<th>Size (MB)</th>
<th>CR</th>
<th>CSR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ImageNet</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EfficientNet-B0</td>
<td>76.43</td>
<td>75.01</td>
<td>21.15</td>
<td>7.62</td>
<td>3.31</td>
</tr>
<tr>
<td>LSQ+ (Bhalgat et al., 2020)</td>
<td>76.43</td>
<td>74.08</td>
<td>21.15</td>
<td>8.25</td>
<td>3.91</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>76.15</td>
<td>75.66</td>
<td>102.23</td>
<td>8.21</td>
<td>3.50</td>
</tr>
<tr>
<td>PWLQ (Fang et al., 2020)†</td>
<td>76.15</td>
<td>75.29</td>
<td>102.23</td>
<td>9.97</td>
<td>4.50</td>
</tr>
<tr>
<td>MMSE (Choukroun et al., 2019)</td>
<td>76.13</td>
<td>75.62</td>
<td>102.23</td>
<td>7.86</td>
<td>2.64</td>
</tr>
<tr>
<td>KURE (shkolnik et al., 2020)†</td>
<td>76.02</td>
<td>74.98</td>
<td>102.23</td>
<td>7.94</td>
<td>2.64</td>
</tr>
<tr>
<td>ResNet-34†</td>
<td>76.30</td>
<td>75.60</td>
<td>102.23</td>
<td>7.88</td>
<td>2.64†</td>
</tr>
<tr>
<td>ResNet-34‡</td>
<td>73.30</td>
<td>72.98</td>
<td>87.19</td>
<td>7.80</td>
<td>4.32</td>
</tr>
<tr>
<td>QIL (Jung et al., 2019)†</td>
<td>73.70</td>
<td>73.70</td>
<td>87.19</td>
<td>6.82</td>
<td>2.65</td>
</tr>
<tr>
<td>DSQ (Gong et al., 2019)‡</td>
<td>73.80</td>
<td>72.76</td>
<td>87.19</td>
<td>6.82</td>
<td>2.65</td>
</tr>
<tr>
<td><strong>CIFAR-10</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResNet-20</td>
<td>91.67</td>
<td>91.60</td>
<td>1.08</td>
<td>8.43</td>
<td>3.92</td>
</tr>
<tr>
<td>SLB (Yang et al., 2020)†</td>
<td>91.67</td>
<td>91.15</td>
<td>1.08</td>
<td>16.23</td>
<td>11.31</td>
</tr>
<tr>
<td>GWS (Zhong et al., 2020)†</td>
<td>92.10</td>
<td>92.10</td>
<td>1.08</td>
<td>7.64</td>
<td>2.62</td>
</tr>
<tr>
<td><strong>MNIST</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeNet-300-100</td>
<td>98.70</td>
<td>98.63</td>
<td>1.07</td>
<td>13.31</td>
<td>7.62</td>
</tr>
<tr>
<td>MLP-GSC</td>
<td>91.00</td>
<td>91.19</td>
<td>2.57</td>
<td>10.88</td>
<td>5.55</td>
</tr>
<tr>
<td>HE (Zhang et al., 2017)</td>
<td>86.40</td>
<td>86.40</td>
<td>0.2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Google Speech Commands</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLP-HGR</td>
<td>88.50</td>
<td>88.33</td>
<td>1.30</td>
<td>8.51</td>
<td>3.96</td>
</tr>
<tr>
<td>HE (Zhang et al., 2017)</td>
<td>88.50</td>
<td>87.22</td>
<td>1.30</td>
<td>13.57</td>
<td>8.35</td>
</tr>
<tr>
<td>HMM (Georgi, Amma, and Schultz, 2015)</td>
<td>74.30</td>
<td>74.30</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

As explained in section 6.4, FantastIC4’s compute system (figure 6.3) is able to compute the output within 1 clock cycle. Thus, its peak compute performance can be calculated as follows

\[
\text{Throughput}_{\text{compute}} = f \times \frac{\# \text{FIFOs} \times 2}{\# \text{FLOPs}} \quad (6.2)
\]
with \( f \) being the clock frequency and \# FIFOs the number of FIFOs, which with the \( \times 2 \) factor corresponds to the number of FLOPs being computed per output element. Thus, for FantastIC4’s 22nm ASIC version running at 800 MHz and with 256 FIFOs, the peak compute performance is 409.6 GOPS.

However, the above measure of throughput does not take into account the number of clock cycles required for loading the elements into the chip, which is often neglected in the literature. Loading data to the compute engine plays a pivotal role in the final performance, and can sometimes become the major bottleneck. In fact, the actual throughput of an accelerator executing a particular algorithm is

\[
\text{Throughput} = \min\{\text{Throughput}_{\text{compute}}, \text{Throughput}_{\text{load}}\}
\]

For instance, consider the parameter matrix \( W \in \mathbb{R}^{m \times n} \) and an activation vector \( A \in \mathbb{R}^n \). When computing \( W \cdot A \) in dense (uncompressed) representation, typically about \( \mathcal{O}(m \times n + n) = \mathcal{O}((m + 1)n) \) clock cycles\(^4\) need to be spend only for loading the values from memory (e.g. a DRAM) to the compute engine. Thus, the throughput of a MVM is fundamentally bounded by the throughput of loading elements, which is determined by the frequency of the accelerator. That is,

\[
\text{Throughput}_{\text{uncompressed}} = \min\{\text{Throughput}_{\text{compute}}, f\}. \tag{6.3}
\]

In contrast, by compressing the parameter and activation values and executing the MVM in their compressed representation, we are directly increasing the effective throughput of loading elements since more data can be loaded per clock cycle. Thus, the resulting throughput increases by the data compression ratio, that is:

\[
\text{Throughput}_{\text{compressed}} = \min\{\text{Throughput}_{\text{compute}}, f \times CR_a \times CR_w\} \tag{6.4}
\]

\[
= f \times \min\{\# \text{ FIFOs} \times 2, CR_a \times CR_w\}. \tag{6.5}
\]

\( CR_a \) takes into account the compression ratio of the activation values, as taken by the ratio between the uncompressed 16bit activation values and the compressed activations. Since we allow for efficient handling of activation values with 16 as well as 8bit values, \( CR_a \in \{1, 2\} \). Analogously, \( CR_w \) denotes the compression ratio of the DNN parameters.

### Benchmarking MLP models

We evaluated FantastIC4’s ASIC version on a 22nm process node with a clock frequency of 800MHz. We benchmarked the end-to-end inference efficiency of our custom and hardware-conform MLPs, trained for the GSC and HGR tasks. Both MLP models, which we named MLP-GSC & MLP-HGR respectively, reach state-of-the-art prediction performance on their tasks (see table 6.2). The total latency to perform the inference was found to be 1.31\( \mu \)s for the MLP-HGR model and 1.37\( \mu \)s for the MLP-GSC model.

Table 6.3 shows the layout, and the area and the power breakdown is shown in the Fig. 6.8. The total area of our processor was found to be 1mm \( \times \) 1.2mm. Most of the area and power consumption is dominated by the adder tree and the FIFOs as it forms the core part of the architecture. This stresses again the benefits of

\(^4\) Assuming 1 clock cycle per read operation, which is typically the case if we consider a 32bit wide memory unit (e.g. a DRAM) for storing the elements.
implementing an ACM computational flow instead of MAC. Namely, an array of 256 MAC units of 16bit-width consumes an area of $346.58 \mu m \times 346.58 \mu m$, whereas the same ACM unit will consume an area of $216.54 \mu m \times 216.54 \mu m$. Similarly, an array of 256 MAC units consumes a power of 101.23mW and an array of 256 ACM units consumes a power of 40.46mW. Thus, our proposed ACM technique saves a total of $\approx 39\%$ in area and $\approx 40\%$ in power. Also notice, how this efficiency breakdown is similar to the computational complexity breakdown results reported in the previous chapter 5.4 for the CER & CSER formats, in that most of the complexity results from performing additions instead of multiplications. Interestingly, we also see that one of the major sources of complexity when processing the sparse format comes from loading the non-zero elements, which is corroborated by the resource consumption utilized by the FIFOs.

When considering the MLP-GSC & MLP-HGR models, we see that FantastIC4’s throughput was dominated by the the clock cycles required for loading data, since as per equation \( 6.5 \): \[ \text{throughput}_{\text{compute}} = 800\text{MHz} \times 256 \times 2 = 409.6 \text{ GOPS} \] \[ \text{throughput}_{\text{load}} = 800\text{MHz} \times 2 \times 13.59 = 21.7 \text{ GOPS} \] However, when compared to uncompressed data, the gains in throughput become directly proportional to the
6.6. Experiments

compression ratio. Thus, FantastIC4’s peak throughput is \times 13.59 higher as compared to uncompressed executions of the aforementioned MLP models.

Comparison to previous work

Arguably, the closest related work to FantastIC4 are EIE (Han et al., 2016) & EyerissV2 (Chen et al., 2019), since both accelerators also leverage compressed representations of the DNN parameters. More recent accelerators exploiting compressed representations do exist, such as (Parashar et al., 2017; Gondimalla et al., 2019), however these were optimized for conv layers, whereas FantastIC4 optimizes the execution of FC layers.

When it comes to throughput for loading data, EIE & EyerissV2 performances also increase proportionally with the data’s compression ratio. However, both accelerators exploit the CSR format which, as explained above, does not always constitute the most compact form of the model parameters. Moreover, EIE works on 16bit precision of the activation values. In contrast, FantastIC4 is able to increase its load throughput thanks to its support for mixed-precision activation values as well as multiple compressed representations of the model parameters. Concretely, for the MLP-GSC model, the compression ratio achieved by FantastIC4 was of 13.59, which is \times 1.7 higher than the CR of EIE & EyerissV2. This directly translates to an increase of \times 3.4 & \times 6.8 in throughput performance for loading data, as per equation (6.4). Again, we stress that, in the case of MLPs, this becomes the major bottleneck when performing inference. Moreover, this also translates to an increase of \times 4.43 & \times 9.03 in load energy efficiency, and of \times 115.25 in terms of load area efficiency, both being paramount when ML models need to be run on power- and area-constrained devices.

In terms of computational throughput, EIE and EyerissV2 scale with the sparsity ratio of the models. That is, the number of clock cycles required to compute an output is proportional to the number of non-zero elements present in the row being processed. Concretely, their effective computational throughput can be estimated by the following formula:

\[
\text{Throughput}_{\text{compute}} = \frac{f \times \#\text{PEs} \times \#\text{FLOPs}}{< \#\text{nnz} >}
\]  

(6.6)

with \#PEs being the number of processor elements, \# FLOPs the number of floating-point operations and \(< \#\text{nnz} > \) the average number of non-zero elements present at each row. If we consider, e.g., the sparsity ratio of 67.65% from the MLP-GSC model (see table D.1 in appendix D for more details), 83 non-zero elements per output would be present on average, considering an input layer size of 256. This would correspond to a throughput of 316.5 GPUs for EIE and of 237.4 GPUs for EyerissV2.

In Table 6.4 we summarize the results. In order to provide a more fair comparison, we also implemented FantastIC4 on a TSMC 65nm LP process. Due to the limitation in the process node, we could operate the architecture to a maximum frequency of 250MHz in order to meet the desired setup timing. This reduced the compute throughput down to 128 GPUs, which lead to worse performance in terms of computational throughput. However, we want to remark that assuming we could

5 Calculated as 256 \times (100 - 67.65)/100.
6 Calculated as 800MHz \times 64 \times 256 \times 2/83
7 Calculated as 200MHz \times 192 \times 256 \times 2/83
TABLE 6.4: Performance comparison with other compression-based ASIC accelerators. Throughputs (GOPS) have been estimated w.r.t. MLP-GSC model.

<table>
<thead>
<tr>
<th>Platform</th>
<th>EIEa</th>
<th>Eyeriss V2b</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology(nm)</td>
<td>65</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>800</td>
<td>200</td>
<td>250</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>590</td>
<td>606</td>
<td>931</td>
</tr>
<tr>
<td>Area (mm²)</td>
<td>40.8</td>
<td>N/A</td>
<td>49</td>
</tr>
<tr>
<td>Compute Throughput (GOPS)</td>
<td>316.5</td>
<td>237.4</td>
<td>128</td>
</tr>
<tr>
<td>Compute Power Efficiency (GOPS/W)</td>
<td>536.4</td>
<td>391.7</td>
<td>137.4</td>
</tr>
<tr>
<td>Compute Area Efficiency (GOPS/mm²)</td>
<td>7.75</td>
<td>N/A</td>
<td>2.6</td>
</tr>
<tr>
<td>Load Throughput (GOPS)</td>
<td>6.4</td>
<td>3.2</td>
<td>6.8</td>
</tr>
<tr>
<td>Load Power Efficiency (GOPS/W)</td>
<td>10.8</td>
<td>5.3</td>
<td>7.3</td>
</tr>
<tr>
<td>Load Area Efficiency (MOPS/mm²)</td>
<td>156.9</td>
<td>N/A</td>
<td>138.8</td>
</tr>
</tbody>
</table>

a (Han et al., 2016). b (Chen et al., 2019)

run our 65nm version at the initial frequency of 800MHz, would outperform the previous work in computational throughput, energy and area efficiency. Moreover, we see that our 65nm version still outperforms the previous work in terms of the data loading throughput by up to \( \times 2.13 \).

When considering our 22nm version, we see that we attain \( \times 1.29 \) & \( \times 1.73 \) gains in computational throughput as compared to EIE & EyerissV2 respectively. Similarly, in terms of computational energy efficiency, we see gains of \( \times 1.68 \) & \( \times 2.3 \) respectively. In terms of computational area efficiency, we see a gain of \( \times 44 \) as compared to EIE, mainly due to the fact that we are able to employ 22nm technology, which drastically reduces the area requirements. Unfortunately, EyerissV2 does not provide explicit area benchmarks, but instead reports it in terms of total number of gates. Hence, by comparing the total gates, we are smaller by \( \times 2.9 \). However, we outperform EIE by \( \times 544 \) when it comes to gate counts.

Finally, we also compare FantastIC4’s efficiency against Guo (Ruiqi Guo et.al, 2019), a RNN-based speech recognition ASIC processor designed specifically for solving key-word spotting (KWS) tasks efficiently. Solving the KWS task at high energy and area efficiency is paramount, since in many real-world use-cases these type of ML models need to run on power- and area-constrained devices (e.g. home assistant devices, mobile phones, etc.). In table 6.5 we see the comparison. Guo’s hardware performance was also benchmarked on the same GSC dataset. Their DNN model attained an accuracy of 90.20% for the task of recognizing 10 keywords, whereas our MLP-GSC model attained 91% for the same task, thus attaining an increase of 0.8% in prediction performance. Moreover, (Ruiqi Guo et.al, 2019) reports a latency of 127.3\( \mu \)s, whereas we were able to run our model at 5.2\( \mu \)s & 1.31\( \mu \)s, thus being able to speedup the task by up to \( \times 97 \). These speedups lead to lower energy consumptions, being able to save up to \( \times 11.32 \) of power (in terms of Joules) for solving the same KWS task at state-of-the-art accuracy. Moreover, we also see gains in area efficiency of up to \( \times 10.91 \), indicating that FantastIC4 is able to utilize the limited resources (area and power) in a more efficient manner.
6.6. Experiments

Table 6.5: Performance comparison with other ASIC accelerators specially designed for the GSC task. Ours are based on the performance of running inference of our custom MLP-GSC model on FantastIC4.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Guo( ^a )</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords Number</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Dataset</td>
<td>GSC</td>
<td>GSC</td>
</tr>
<tr>
<td>Accuracy</td>
<td>90.20%</td>
<td>91%</td>
</tr>
<tr>
<td>Technology (nm)</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>75</td>
<td>250</td>
</tr>
<tr>
<td>Latency (( \mu s ))</td>
<td>127.3</td>
<td>5.2</td>
</tr>
<tr>
<td>Area (mm(^2))</td>
<td>6.2</td>
<td>49</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>52.5</td>
<td>931</td>
</tr>
<tr>
<td>Energy (( \mu J ))</td>
<td>6.68</td>
<td>4.84</td>
</tr>
<tr>
<td>Area Efficiency (nJ/mm(^2))</td>
<td>1077.94</td>
<td>98.8</td>
</tr>
</tbody>
</table>

\( ^a \) (Ruiqi Guo et al., 2019).

Figure 6.9: Power consumption of our MLP-HGR model as a function of its entropy distribution. (Blue) Dynamic power consumption measured on an FPGA, (Red) measured on ASIC simulation ©2021 IEEE.

6.6.4 Ablation studies

In section 6.3 we argued that one of our major contributions is the fact that FantastIC4’s hardware architecture is specially designed to exploit low-entropy statistics of the weight parameters. Thus, we should expect the execution efficiency of DNN models to increase as the entropy of the weight parameters decreases. Figure 6.9 shows exactly this trend. In this experiment, we present the power-efficiency of our MLP-HGR model at different overall entropy levels of the model. To perform
this study, we ran our post-layout simulation (ASIC) and post-implementation timing simulation (FPGA) to generate the corresponding Value Change Dump (VCD) for ASIC and Switching Activity Interchange Format (SAIF) for FPGA. Using these files we measured the dynamic (Vector Based) power consumption on the Synopsys PrimeTime and Vivado. Based on the measurement, the power consumption decreases quasi linearly with the entropy of the model. Again, this trend is due to the fact that FantastIC4 supports (1) the efficient processing of compressed representations of the weight parameters, (2) efficient computation of 4bit non-zero values, and (3) efficient loading of repeated values from the FIFOs; all being properties that become more and more predominant as the entropy of the models parameters decreases. For more experiments and ablation studies, we refer to the appendix D.

6.7 Summary

In this chapter, we describe a hardware-software co-design principle for obtaining an efficient execution engine of deep neural networks (DNNs) that are based on FC layers. We introduce several optimization techniques aiming to maximally increase the area and power efficiency for performing inference. Firstly, FantastIC4 supports the efficient on-chip execution of multiple compact representations of fully-connected layers, as such boosting the data movement and area efficiency of the DNNs parameters. It also implements a first-accumulate-then-multiply (ACM) computational paradigm in order to (1) increase the computational efficiency by drastically reducing the number of multipliers down to 4 (hence the name FantastIC4) and (2) the on-chip data movement efficiency of the activation values.

We evaluated FantastIC4’s performance on an ASIC version, synthesized on a 65nm & 22nm process node with a clock frequency of 250MHz & 800MHz respectively. Our experiments show that we were able to reach a compute and load efficiency of up to 902.2 GOPS/W and 47.9 GOPS/W respectively, thus being up to \( \times 9.03 \) more efficient than previous work. In terms of area efficiency, FantastIC4 showed gains of up to \( \times 115 \) as compared to previous work. Evaluated on the Google Speech Command dataset, we showed that we are able to solve the respective keyword spotting task at a latency of only \( 1.31 \mu s \), which was a significant speedup of \( \times 97 \) as compared to previous work. These speedups also lead to \( \times 11.32 \) savings in power consumption and an increase of \( \times 10.91 \) in area efficiency, both gains being of utmost importance for use-cases where the ML models need to be executed on resource-constrained devices with limited area and power budget.

6.8 Limitations

A natural limitation of FantastIC4’s hardware architecture is that, in its current form, it can only process FC layers. In order to be able to support other type of layers such as convolutions or self-attention layers, optimizations regarding the data movement of the activation values would have to be considered as well.

A further limitation is that FantastIC4 can only process DNN parameters that can be represented as a linear combination of 4 basis weights. Although our introduced 4-bit entropy-constrained training method make DNNs more amenable to 16-bit quantization, the constrain may still be too strong for high complex tasks. In such cases, quantizing the parameters down to 16-bits may significantly impact the prediction performance of the models, even after retraining.
Chapter 7

Conclusion

The past decade has witnessed a surge of AI-based technologies, disrupting many industries and impacting large sectors of our society. This fast progress was fueled by several breakthroughs achieved thanks to the research conducted on deep neural networks. Many of these breakthroughs were possible thanks to finding methods that allow to train larger and larger DNN models, with hundreds of millions or even billions of parameters. However, this came at the price of having to spend enormous amounts of resources for training as well as executing the models in terms of memory, speed, energy and economical costs.

In this thesis we discussed methods for reducing the memory and inference costs of DNN models, by means of reducing the information content of their parameter set. Concretely, one of the main focus lied around the question of

\[ \text{how can we reduce the first-order entropy statistics of the parameters and, subsequently, leverage this property in order to increase the memory and compute efficiency of the models?} \]

Each chapter is centred around providing a particular contribution to the above question.

**Entropy-constrained training (chapter 3):** Firstly, we characterized the information content of DNN models as being bounded by the first-order entropy of their parameter set, defined as the entropy of the empirical probability mass distribution. We then formalized an entropy-constrained training objective, and described manners in which we can optimize it by applying scalable, gradient-based techniques. Our experimental results showed that we can reduce the information content of popular image classification models by up to \( \times 63 \), while reducing the required number of arithmetic operations (multiplications and summations) for performing inference by up to \( \times 60 \). With our entropy-constrained training method we were able to design highly compact and efficient models, leading us to be among the top 5 best submissions in the NeurIPS2019 MicroNet Challenge\(^1\), a competition that had as one of its goals incentivizing the development of efficient models and model compression techniques.

**Compact representations of DNNs (chapters 4 & 5):** Secondly, we focused on designing compression algorithms that encode the models information into compact representations. Here we considered two types of representation, one that outputs a maximally compressed representation but is not amenable for inference (highest memory but low computational efficiency, chapter 4), and one that is compact and reduces the number of operations for performing inference at the same time (lower memory but higher computational efficiency, chapter 5).

\(^1\)https://micronet-challenge.github.io
• **DeepCABAC.** In chapter 4 we described DeepCABAC, a universal compression engine for DNNs tailored to produce maximally compact representations of them. It is based on applying Context-based Adaptive Binary Arithmetic Coding (CABAC) to the parameters of the models, which is a state-of-the-art lossless compression algorithm employed in popular media compression standards such as in H.264 & H.265. In our experimental results we showed that DeepCABAC is able to compress the models by up to $\times 51$ on average, attaining a model with 75.03% accuracy but merely 1.31MB in size on the popular ImagNet classification task. As of today, DeepCABAC is the state-of-the-art compression technology for DNNs in the literature and industry, and it has been selected as the core technology in the upcoming MPEG-7 standard for NN compression (Kirchhoffer et al., 2021).

• **CER & CSER.** In chapter 5 we designed two novel matrix formats that produce compact representations of the tensor, that at the same time require less number of operations for performing the dot product. Their design are based on applying run-length coding techniques akin to sparse formats such as the CSR, but generalized so that they can exploit better the low-entropy statistics of the parameters. In addition, the CER & CSER formats leverage the distributive law in order to significantly reduce the required number of multiplications that need to be performed for inference. Experimental results show that we can attain up to times 41.95 compression gains on large models like VGG16, and a reduction of their computational complexity by up to $\times 5.53$, thus attaining $\times 2.5$ improvements in compression and $\times 1.5$ higher computational efficiency than previous alternatives.

**Specialized hardware architecture (chapter 6):** Finally, we introduced FantastIC4, a hardware architecture that is tailored to execute compact representations of DNNs in a highly efficient manner. FantastIC4 incorporates similar optimization paradigms as in chapter 5, in that it leverages the distributive law of multiplications in order to reduce the total number of multiplications down to only 4 per output element (hence the name). This lead to a $\times 2.7$ improvement in the energy consumption spend in performing computations. It also supports the efficient on-chip processing of multiple sparse compressed representations, which lead to a boost of $\times 2$ in compression gains on average, as such reducing further the area requirements as well as the energy consumption spend from off- to on-chip data movement. Our experiments showed that the ASIC version achieved a computational and load power efficiency of up to 902.2 GOPS/W & 47.9 GOPS/W respectively, consequently increasing the area efficiency by up to $\times 115$ as compared to previous work. Moreover, when compared to other, state-of-art accelerators designed for efficiently solving keyword-spotting tasks, it attained speedups of $\times 97$, which lead to $\times 11.32$ savings in power consumption.

All in all, this thesis focused on applying information theoretical principles and techniques in order to reduce the memory as well as computational complexity of DNN models at inference time.

**Future work:** However, the amount of resources demanded by training also constitutes one of the big caveats of DNN models, specially since the processing of big-data is also required in the process (LeCun et al., 2012). Similarly to inference, one can try to reduce the memory as well as computational complexity of training by applying
techniques such as pruning, quantization and/or lossless compression. For instance, in one of our work we were able to reduce the number of computations required for training by 92% on average, by applying stochastic sparsification techniques (Wiedemann et al., 2020b). In another work we reduce the communication costs required in distributed training scenarios by four orders of magnitude, by applying sparsification, quantization and lossless compression to the parameter updates (Sattler et al., 2019; Sattler et al., 2020). Hence, there is great potential in being able to further reduce the resource requirements for training by means of information theoretical principles and techniques akin to the ones presented in this thesis. Moreover, specialized hardware architectures supporting special type of optimization techniques tailored to reduce training costs is still an active topic of research.

Furthermore, some of the here studied techniques may also be applied in order to accelerate other type of ML algorithms, such as kernel-based classification and regression models (Müller et al., 2001; Braun, Buhmann, and Müller, 2008).

As the size of DL models increases over time, progress along the efficient processing of DNNs will become increasingly relevant for their practical usage. We hope this thesis motivated researchers and engineers to further study this topic and to help develop practical solutions to the problem. Again, as in the words of (Thompson et al., 2020), continued progress in DL-based applications will require dramatically more computationally-efficient methods, which will either have to come from changes to deep learning or from moving to other machine learning methods.
Appendix A

Chapter 3 - Supplemental material

A.1 Proof of theorem 1

Firstly, we refresh some of the definitions and notations introduced in chapter 3.

We denote with $W \in \mathbb{R}^n$ a particular set of parameter values of the DNN, with each element $W_i$ admitting only values from a finite set $\Omega$, thus $W_i \in \Omega = \{ \omega_0, ..., \omega_{K-1} \}$. We denote with $K = |\Omega|$ the cardinality of $\Omega$, and with $\mu$ the EPMD of the set $\Omega$, as measure by $W$, thus $\mu_k = \#(\omega_k)/n$.

In the context of theorem 1, we defined the probability of a parameter $w_i \in \mathbb{R}$ sampling a discrete value $\omega_k$ as follows

$$ P_{ik} = \frac{G_{\sigma}(w_i, \omega_k)}{Z_i}, $$

with $G_{\sigma}(w_i, \omega_k) = e^{-\frac{1}{2}(\frac{w_i - \omega_k}{\sigma})^2}$

and $Z_i = \sum_k G_{\sigma}(w_i, \omega_k)$

Furthermore, we defined the continuous relaxation of the EPMD and the respective entropy as to be

$$ P_k = \frac{1}{n} \sum_i P_{ik} $$

$$ H(P) = - \sum_k P_k \log_2 P_k $$

Let $\hat{W} \in \mathbb{R}^n$ denote the continuous parameter values $w_i$ entailed in the definition of $P_{ik}$ above. Then, we define the variational loss function as to be

$$ \mathcal{L}(P_\theta) = \mathbb{E}_{P_\theta} [ - \log_2 p(Y|X, W \sim P_\theta)] + \alpha nH(P(\theta)) $$

with $\theta = (\hat{W}, \Omega, \sigma)$ composed by the set of continuous parameters $\hat{W}$, the discrete set $\Omega$ and the standard deviation of the gaussian kernel $\sigma$. Finally, we define the respective discrete loss function as

$$ \mathcal{L}(W) = - \log_2 p(Y|X, W) + \alpha nH(\mu), $$

with $W = q(\theta)$ being the MAP estimate of $P(\theta)$.

We also state again the theorem
Theorem 1 (Loss upper bound): Let the probability $P_{ik}$ of a weight parameter $w_i$ sampling the discrete value $\omega_k$ be defined by the gaussian kernel as follows

$$P_{ik} = \frac{G_\sigma(w_i, \omega_k)}{Z_i},$$

with $G_\sigma(w_i, \omega_k) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2}(w_i - \omega_k)^2}$

and $Z_i = \sum_k G_\sigma(w_i, \omega_k)$

Let further the probability $P_k$, and the respective entropy $H(P)$ be defined as in (3.5). Finally, let $\theta = (W, \Omega, \sigma)$ denote the set of parameters parametrizing the probability distributions $P = \{P_k | k \in \{0, \ldots, K-1\}\}$, with $\theta^*$ minimizing the variational objective (3.6), and with $W^* = q(\theta^*)$, $\mu^*$, being the respective MAP point hypothesis and respective EPMD. Then, for sufficiently small variance of the distribution $P_{\theta^*}$, and assuming that the loss function is locally convex around the minima, the following relationship holds

$$L(P_{\theta^*}) = E_{P_{\theta^*}} [-\log_2 p(Y|X, W \sim P_{\theta^*})] + anH(P(\theta^*))$$

$$> L(W^*) = -\log_2 p(Y|X, W^*) + anH(\mu^*)$$

A.1.1 Proof

Lemma 1. Let $H(P(\theta))$ and $H(\mu)$ be the entropy as defined in theorem 1. Then, the following inequality

$$H(P) \geq H(\mu)$$

holds.

Proof. Firstly, notice how $P_{ik} \xrightarrow{\sigma \to \infty} 1/K$. That is, the continuous relaxation $P_{ik}$ tends to become the uniform distribution over $K$ elements, as the variance of the gaussian kernel increases. This implies that, for any $\sigma > 0$, $P_k = \frac{1}{K} \sum_k P_{ik}$ gets monotonically “closer” to the uniform distribution. Consequently, the entropy $H(P)$ increases monotonically with $\sigma$.

In addition, notice how, analogously, $P_{ik} \xrightarrow{\sigma \to 0} 1$ for $k = \min_k(w_i - \omega_k)$, and $P_{ik} \xrightarrow{\sigma \to 0} 0$ for the rest. This implies that $P_k \xrightarrow{\sigma \to 0} \mu_k$.

Consequently, $H(\mu)$ must be smaller than $H(P)$ for all $\sigma > 0$. \(\square\)

Lemma 2. Let $\theta^*$ be the minima of the variational loss. For sufficiently small variance $\sigma$, and assuming the loss function is locally convex, then

$$L(P_{\theta^*}) = E_{P_{\theta^*}} [-\log_2 p(Y|X, W \sim P_{\theta^*})]$$

$$> L(W^*) = -\log_2 p(Y|X, W)$$

Proof. By following the same paradigm as in lemma 1, we can infer that $P(\theta)$ will most likely sample values close to the MAP estimate $W = q(\theta)$ if $\sigma \to 0$.

We can then expand the variational loss around $W$ as follows: Let $e = (\hat{W} - W^*) \sim P(\theta^*)$ represent the error samples around $W^*$ on the minima $\theta^*$, then

$$L(P_{\theta^*}) = L(W^*) + \partial_W L(W) E_{P_{\theta^*}} [e] + \partial^2_W L(W) E_{P_{\theta^*}} [e^2] + O(E_{P_{\theta^*}} [e^3])$$

$$= 0$$
A.1. Proof of theorem 1

Assuming the Hessian $\frac{\partial^2}{\partial W^2} \mathcal{L}(W)$ is positive semidefinite, then it trivially follows that $\mathcal{L}(P^*) > \mathcal{L}(W^*)$.

Proof of theorem. From lemmas 1 and 2, the proof of theorem 1 trivially follows.

A.1.2 Experimental validation of theorem 1

Figures A.1a, A.1b and A.2 verify some of the arguments stated in the proof of theorem 1. The experimental setup is as described in chapter 3, section 3.2.4.

Firstly, figure A.1a shows how the entropy of the quantized model is always lower than the entropy of the continuous parametrization. Thus, the entropy of the continuous relaxation is always upper bounding the entropy of the quantized model. We also confirmed this on the other architectures.

Secondly, from figure A.2 we see how the loss automatically tends to minimize the variance of the continuous probability distribution during training. This increases the probability of selecting the quantized model (or close models) during training and, thus, minimizing its loss value as well.
Lastly, from figure A.1b we see how indeed, this can result in the variational classification loss eventually bounding the loss of the quantized network. We verified a similar trend on the LeNet-300-100 architecture. In particular, as we predicted, by comparing figures A.2 and A.1b we see how the difference between the variational and the quantized loss converges to the same value, as the variance of the continuous probability distribution becomes smaller.

A.2 Details on the experimental setup of section 3.3.4

Firstly, as mentioned in the main text, we want to stress once more that the code and implementation of the experiments are publicly available under https://github.com/d-becking/efficientCNNs.

A.2.1 Datasets

As stated in table 3.2, we benchmarked our models on the CIFAR and ImageNet datasets.

The two CIFAR (Krizhevsky, 2009) datasets consist of natural images with a resolution of 32 × 32 pixels. CIFAR-10 contains 10 classes and CIFAR-100 100 classes. The train and test sets contain 50,000 and 10,000 images.

Disclaimer: Parts of section A.2 are reprinted and with permission from (Marban et al., 2020; Becking, 2020).
In contrast, the ImageNet (Russakovsky et al., 2015) dataset is a large-scale dataset containing 1.2 million training images and 50,000 test images of 1000 classes. The resolution of the image data is various and in the range of several hundred pixels. In DNN applications, the ImageNet data is usually cropped to 224 × 224 pixels.

A.2.2 Regularization techniques

We used Dropout (Srivastava et al., 2014) in order to partially regularize the training of the models. Concretely, in all our CIFAR models we used a dropout rate of 0.2 between the two convolutional layers into each residual block. In EfficientNet, we also used a dropout rate of 0.2 is applied but only to the final fully connected layer.

We also apply data augmentation techniques. We crop and randomly flip horizontally the samples of all datasets. For CIFAR-10 and CIFAR-100, we additionally apply cutout as described in (Devries and Taylor, 2017). Moreover, we also use PyTorch’s ColorJitter which alters brightness, contrast, saturation and hue depending on a given probability. Prior to data augmentation we normalize the input data.

Finally, we also apply L2-norm regularization to the parameter values while training. For the CIFAR nets, we use a weight decay of 5 × 10^{-4}, for EfficientNet 1 × 10^{-4}. to the non-quantized parameters, i.e. the input layer, BatchNorm layers, etc., we apply a weight decay of 5 × 10^{-6}.

A.2.3 Training Procedure and Optimization

We employed ADAM (Kingma and Ba, 2015) as the optimizer while applying the ternarization process EC2T described in (Marban et al., 2020) to the networks. We set the learning rate to 1 × 10^{-4} for updating the weight parameters \( w_i \) and to 1 × 10^{-5} for updating the centroids \( \omega_k \).

We train the ternary assignment plus the centroid values for 20 epochs. Afterwards, we fix the assignment and fine-tune the centroid values only for further 15 epochs. In the CIFAR networks, we apply EC2T to all convolution layers but the first. In order to compress EfficientNet-B1, we apply EC2T to the expansion, projection and convolutional head layers. Afterwards we apply entropy-constrained training to the depthwise convolutions, the squeeze-and-excitation convolutions and the fully connected layer with 15 epochs each, while keeping the ternarized and pruned layers fixed. In a final step, we fix all compressed layers and retrain the remaining parameters for 15 epochs.
Appendix B

Chapter 4 - Supplemental material

B.1 Experiment details

B.1.1 Uniform quantization

Uniform quantization is essentially one step of the weighted Lloyd algorithm with no importance measure, $\lambda = 0$, and no cluster center update. One major difference between uniform quantization and the weighted Lloyd algorithm is that in the weighted Lloyd algorithm, the neural network is quantized as a whole, while in uniform quantization, the neural network is quantized layer-wise (see Algorithm 8).

For the experiment described in section V-A, the number of clusters were determined by first starting out with 256 clusters for un-sparsified networks and with 32 clusters for sparsified networks. Then the networks were quantized and evaluated. If the accuracy was not within a range of $\pm 0.5$ percentage points as compared to the original accuracy, then the number of clusters was doubled until the accuracy was within that range.

The quantized models were then compressed using scalar Huffman coding and bzip2. Additionally, the sparse models were compressed using CSR-Huffman coding. Since additional parameters such as biases were not quantized, their original size was added to the compressed size of all compression methods.

B.1.2 Lloyds hyperparameter selection

To determine the optimal number of clusters and optimal values for $\lambda$, in the beginning, the number of clusters were fixed to 256. The ranges for $\lambda$ were determined iteratively under the assumption that the accuracy decreases roughly monotonic with an increasing $\lambda$. At first, 20 experiments with a $\lambda$ value between 0.0 and 1.0 were started and evaluated to establish a rough range of $\lambda$ in which the accuracies are within a range of $\pm 0.5$ percentage points as compared to its original accuracy. In one case (LeNet-300-100 Sparse) all experiments yielded accuracies within that range. In that case another 20 experiments were conducted with $\lambda$ values between 1.0 and 2.0. In the case of MobileNet v1 and its sparse counterpart, even a value of 0.0 for $\lambda$ produced accuracies below the $\pm 0.5$ percentage point threshold. In both cases the number of clusters was doubled, then 20 experiments with $\lambda$ in the range of 0.0 to 1.0 were conducted. This process was repeated until the accuracies were within the threshold$^1$. Then for all networks two adjacent values of $\lambda$ were selected where the accuracy was within the range and that produced the smallest entropies. Again, 20 experiments with $\lambda$ values between these selected $\lambda$s were conducted. This process

---

$^1$Please note that, although the number of clusters was drastically increased for MobileNet v1, we were not able to achieve accuracies within the target threshold.
Table B.1: The number of clusters per layer employed for the experiment described in the experimental section V-A.

<table>
<thead>
<tr>
<th>Model</th>
<th>Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeNet-300-100</td>
<td>256</td>
</tr>
<tr>
<td>LeNet-300-100 Sparse</td>
<td>32</td>
</tr>
<tr>
<td>LeNet5</td>
<td>256</td>
</tr>
<tr>
<td>LeNet5 Sparse</td>
<td>32</td>
</tr>
<tr>
<td>Small-VGG16</td>
<td>256</td>
</tr>
<tr>
<td>Small-VGG16 Sparse</td>
<td>128</td>
</tr>
<tr>
<td>ResNet50</td>
<td>256</td>
</tr>
<tr>
<td>ResNet50 Sparse</td>
<td>256</td>
</tr>
<tr>
<td>VGG16</td>
<td>256</td>
</tr>
<tr>
<td>VGG16 Sparse</td>
<td>256</td>
</tr>
<tr>
<td>MobileNet v1</td>
<td>1,024</td>
</tr>
<tr>
<td>MobileNet v1 Sparse</td>
<td>1,024</td>
</tr>
</tbody>
</table>

was repeated until there were no longer any gains in the entropies. Typically just two rounds were enough to find no further improvement.

B.1.3 DeepCABAC’s hyperparameter selection

In all experiments we set the $\text{AbsGr}(n)$-Flag to 10.

**DC-v1**

For the experiment in section V-A, we searched through the following set of hyperparameters:

$$S = [0.0, 8.0, 16.0, 32.0, 64.0, 96.0, 128.0, 160.0, 172.0, 192.0, 256.0],$$

$$\lambda = 0.0001 \cdot 2^{\left(\log_2 \frac{0.15}{\imath}\right)}, \quad \forall \imath \in \{0, ..., 99\}$$

**DC-v2**

For the experiment in section V-A we searched through the following set of hyperparameters:

$$\lambda = \frac{0.02}{20} \cdot \imath + 0.01, \quad \forall \imath \in \{0, ..., 20\}$$

$$\Delta_1 = 0.001 \cdot 2^{\left(\log_2 \frac{0.15}{\imath}\right)}, \quad \forall \imath \in \{0, ..., 70\}$$

$$\Delta_2 = 0.064 \cdot 2^{\left(\log_2 \frac{0.128}{\imath}\right)}, \quad \forall \imath \in \{0, ..., 30\}$$

B.2 Approximations to the Fisher Information Matrix

The authors of (Molchanov, Ashukha, and Vetrov, 2017) proposed a sparsification algorithm for neural networks that is based on the minimization of a variational...
B.2. Approximations to the Fisher Information Matrix

<table>
<thead>
<tr>
<th>Model</th>
<th>λ</th>
<th>Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeNet-300-100</td>
<td>0.0144</td>
<td>256</td>
</tr>
<tr>
<td>LeNet-300-100 Sparse</td>
<td>1.1053</td>
<td>256</td>
</tr>
<tr>
<td>LeNet5</td>
<td>0.1222</td>
<td>256</td>
</tr>
<tr>
<td>LeNet5 Sparse</td>
<td>0.4</td>
<td>256</td>
</tr>
<tr>
<td>Small-VGG16</td>
<td>0.2105</td>
<td>256</td>
</tr>
<tr>
<td>Small-VGG16 Sparse</td>
<td>0.2368</td>
<td>256</td>
</tr>
<tr>
<td>ResNet50</td>
<td>0.05</td>
<td>256</td>
</tr>
<tr>
<td>ResNet50 Sparse</td>
<td>0.0105</td>
<td>256</td>
</tr>
<tr>
<td>VGG16</td>
<td>0.0063</td>
<td>256</td>
</tr>
<tr>
<td>VGG16 Sparse</td>
<td>0.0474</td>
<td>256</td>
</tr>
<tr>
<td>MobileNet v1</td>
<td>0.9474</td>
<td>10,240</td>
</tr>
<tr>
<td>MobileNet v1 Sparse</td>
<td>0.9474</td>
<td>3,072</td>
</tr>
</tbody>
</table>

objective. Concretely, they assume the improper log-scale uniform prior \( P(w) \) and assume a fully factorized gaussian posterior over the weight parameters \( P(w|\mu, \sigma) \), and minimize the corresponding variational upper bound

\[
(\mu, \sigma)^* = \min_{(\mu, \sigma)} \mathbb{E}_{P(w|\mu, \sigma)} \left[ \mathcal{L}(y, y') \right] + \beta D_{KL}(P(w|\mu, \sigma)||P(w)) \tag{B.1}
\]

with \( y' \sim P(y'|x, w) \) being the output samples of the neural network, \((x, y)\) the data samples, and \((\mu, \sigma)\) the mean and standard deviations of all the networks parameters, and \( \beta \in \mathbb{R} \) the Lagrange-multiplier. As the KL-Divergence cannot be calculated analytically, they proposed to approximate it by

\[
D_{KL}(P(w|\mu, \sigma)||P(w)) \approx \sum_i k_1 \text{sgm}(k_2 + k_3 \log \alpha_i) - \frac{1}{2} \log \left( \frac{\alpha_i + 1}{\alpha_i} \right) \tag{B.2}
\]

with \( \text{sgm}(\cdot) \) being the sigmoid function, \( \alpha_i = \sigma_i^2 / \mu_i^2 \) the inverse of the signal-to-noise ratio of the parameters, \( k_1 = 0.63576, k_2 = 1.87320 \) and \( k_3 = 1.48695 \). Then, they minimize (B.1) by applying scalable sampling techniques proposed by (Kingma, Salimans, and Welling, 2015).

B.2.1 Connection between pruning and quantization

As a result of minimizing (B.1) we get a mean and standard deviation for each parameter of the network. In our work, we interpreted the former as the (new) value and the latter as a measure of the “robustness” against perturbations. Indeed, the authors suggested to prune away (set to 0) parameters with a signal-to-noise-ratio under a given threshold. Concretely, they suggested the following pruning scheme

\[
w_i \to 0, \quad \forall i : \alpha_i^{-1} < e^{-3}
\]
where \( w_i \equiv \mu_i \) represents now the mean value and thus \( \alpha^{-1} = w_i^2 / \sigma_i^2 \). We can see that the scalar rate-distortion objective
\[
(Q, Q^{-1})^* = \min_{(Q, Q^{-1})} F_i (q_i - w_i)^2 + \lambda L_Q(b)
\]
is a generalization of the above sparsification scheme. Namely, if we assume that the set of quantization points entails the same elements as the input set \( W = Q \) (thus, \( Q^{-1} \equiv \text{identity map} \)), and consider a decoder that assumes a spike-and-slab distribution over the quantization points, then the above objective can be solved by applying the Lloyd algorithm. After convergence, it results in the following solution
\[
Q^*(w_i) = 0 \quad \text{if} \quad F_i w_i^2 < \lambda (b + \log_2 p_0),
\]
with \( p_0 \) being the empirical probability distribution of the 0 value and \( b \) the bit-precision for representing the non-zero values. Hence, if we now choose \( F_i = 1 / \sigma_i^2 \) and the adequate \( \lambda \), we get the suggested criteria as a special case solution. This insight motivated our choice of FIM-diagonals in our experimental section.

### B.2.2 Connection between variances, Hessian, and FIM-diagonals

Firstly, as thoroughly discussed in (Martens, 2020) and mentioned in (Achille, Rovere, and Soatto, 2019), it is important to recall that the FIM is a semi-positive approximation of the Hessian of the loss function. Hence, similar to (Achille, Rovere, and Soatto, 2019), we can derive a more rigorous connection between the estimated variances from minimizing (B.1), the FIM-diagonals and the Hessian. Namely, assuming that the variational loss function can be approximated by its second order expansion around the weight configuration \( w \), we get the following expression
\[
\mathbb{E}_{P(w|\mu, \sigma^2)} [\mathcal{L}(y, y')] \approx \mathcal{L}(w) + \frac{1}{2} \text{tr}[\sigma H(w)]
\]
with \( \mathcal{L}(w) \) being the loss value at \( w \) and \( \text{tr}[\cdot] \) the trace. Hence, if we substitute this expression into (B.1) and take the derivative with respect to \( \sigma_i^2 \) we attain
\[
H_i = \frac{\beta}{\sigma_i^2} K(\alpha^{-1})
\]
with \( K(\alpha^{-1}) \in (0, 1) \) being (approximately) a monotonically increasing function of the signal-to-noise ratio of the parameter. Hence, there is a direct connection between the variances, signal-to-noise ratio, and hessian of the loss function and, consequently, with the FIM-diagonals.

### B.2.3 Hessian-weighted vs. variance-weighted quantization

The authors of (Choi, El-Khamy, and Lee, 2017) suggested a Hessian-weighted Lloyd algorithm for quantizing the neural network parameters, where the diagonals of the empirical Hessian are taken as weights in the algorithm. As we have already discussed above, these coefficients are closely connected to the FIM-diagonals, and are thus also theoretically well motivated. However, we experienced the algorithm to be less stable in practice when we used the Hessian-diagonals instead of the variances. Figure B.1 shows the rate-accuracy curves when we quantized the LeNet5 model.
with both alternatives. As we can see, the curves of the variances are more stable, even achieving better compression results than the Hessian-weighted variant.

B.2.4 More details on the experiments performed in chapter 4

As explained in the above subsections, after minimizing (B.1) we attain a mean and a variance for each parameter of the neural network, where the former can be interpreted as its (new) parameter value and the latter as an estimation of the FIM-diagonals. Moreover, as shown in (Molchanov, Ashukha, and Vetrov, 2017), one can attain highly sparse neural networks as a result from the minimization procedure. Therefore, we applied this sparsification technique to pretrained MobileNetv1, Small-VGG16 and to the LeNet models. However, due to the high memory and computational complexity required in the minimization process, we instead decided to sparsify the VGG16 and ResNet50 networks by applying the iterative algorithm (Han et al., 2015). Subsequently, we estimated their FIM-diagonals by minimizing the objective (B.1) only for the variances, thus, fixing the parameter values during the minimization process. We analogously estimated the FIM-diagonals of the pretrained (non-sparse) versions of the same models.

B.3 Pseudocodes
Algorithm 4 Encoding a message using a Huffman code.

1: Input: A message $M := (s_1, s_2, \ldots, s_n)$ of length $n$, over alphabet $\Sigma$ and a Huffman code $H : \Sigma \to \{0, 1\}^+$
2: Output: A sequence of bits $E \in \{0, 1\}^+$ containing the encoded symbol sequence
3: Let $E = \epsilon$
4: for all $s \in M$ do
5: $E \leftarrow E \circ H_s$
6: return $E$

Algorithm 5 Decoding a message using a Huffman code.

1: Input: A sequence of bits $E \in \{0, 1\}^+$ to decode and a Huffman code $H : \Sigma \to \{0, 1\}^+$
2: Output: A message $M := (s_1, s_2, \ldots, s_n)$ of length $n$, over alphabet $\Sigma$ representing the decoded message
3: Let $M = \epsilon$
4: while $|E| > 0$ do
5: for all $\sigma \in \Sigma$ do
6: if $\exists x \in \Sigma^* : E = H_\sigma \circ x$ then
7: $M \leftarrow M \circ \sigma$
8: $E \leftarrow (E_{|E|}, \ldots, E_{|E|-|E_\sigma|})$
9: return $M$

Algorithm 6 Generating a Huffman code for a message

1: Input: A message $M := (s_1, s_2, \ldots, s_n)$ of length $n$, over alphabet $\Sigma$
2: Output: A huffman code, which maps symbols to code words: $H : \Sigma \to \{0, 1\}^+$
3: for all $\sigma \in \Sigma$ do
4: $F_\sigma \leftarrow \frac{|\{s \in M \mid s = \sigma\}|}{|M|}$ \hspace{1cm} \text{▷ Calculate symbol frequency}
5: Let $Q$ be a priority queue \hspace{1cm} \text{▷ Dequeueing from $Q$ yields smallest element}
6: for all $s \in M$ do
7: if $F_s > 0$ then
8: \text{ENQUEUE}((s, F_s), Q)
9: while $|Q| > 1$ do \hspace{1cm} \text{▷ Build Huffman tree}
10: $(n_l, F_l) \leftarrow \text{DEQUEUE}(Q)$
11: $(n_r, F_r) \leftarrow \text{DEQUEUE}(Q)$
12: \text{ENQUEUE}(((n_l, n_r), F_l + F_r), Q)
13: $(n, F_n) \leftarrow \text{DEQUEUE}(Q)$
14: return $\text{ASSIGNCODEWORDS}(n, \epsilon)$
15: procedure $\text{ASSIGNCODEWORDS}(n, c)$
16: if $n$ is leaf node then \hspace{1cm} \text{▷ Assign code word to leaf nodes}
17: $H_0 \leftarrow c$
18: else \hspace{1cm} \text{▷ Traverse tree recursively}
19: $H \leftarrow H \cup \text{ASSIGNCODEWORDS}(n_l, c \circ 0)$
20: $H \leftarrow H \cup \text{ASSIGNCODEWORDS}(n_r, c \circ 1)$
21: return $H$
Algorithm 7 Weighted Lloyd’s algorithm

1: Input: A set of clusters \( C = \{c_1, c_2, \ldots, c_k\} \), a set of neural network parameters \( W = \{w_1, w_2, \ldots, w_n\} \), an importance measure for each neural network parameter \( F = \{f_1, f_2, \ldots, f_n\} \), and a Lagrangian multiplier \( \lambda \)

2: Output: A set of cluster assignments \( C_j = \{w_1, w_2, \ldots, w_i\} \forall j \in \{1, \ldots, k\} \) that minimize the Lagrangian loss function \( J_\lambda \)

3: for \( j = 1 \to k \) do ▷ Initialize cluster probabilities
4: \( P_j \leftarrow \frac{1}{k} \)
5: repeat
6: for \( j = 1 \to k \) do ▷ Reset the clusters
7: \( C_j \leftarrow \emptyset \)
8: for \( i = 1 \to n \) do ▷ Assignment step
9: \( j = \arg\min_j \left( F_i(W_i - C_j)^2 - \lambda \log_2(P_j) \right) \)
10: \( C_j \leftarrow C_j \cup \{W_i\} \)
11: for \( j = 1 \to k \) do ▷ Update step
12: \( C_j \leftarrow \frac{\sum_{W_i \in C_j} F_i W_i}{\sum_{W_i \in C_j} F_i} \)
13: \( P_j \leftarrow \frac{|C_j|}{n} \)
14: \( j = \arg\min_j |C_j| \) ▷ Enforce 0-cluster
15: \( C_j \leftarrow 0 \)
16: \( J_\lambda \leftarrow \sum_{i=1}^k \sum_{W_i \in C_j} F_i(W_i - C_j)^2 - \lambda \log_2(P_j) \) ▷ Calculate loss
17: until convergence, e.g. \( J_\lambda \) decreases below some threshold
18: return \( C \)

Algorithm 8 Uniform quantization algorithm

1: Input: A set of \( n \) neural network layers \( W_l = \{w_1, w_2, \ldots, w_{n_l}\} \forall l \in \{1, \ldots, n\} \) and a set of cluster centers for each layer of the neural network: \( C_l = \{c_1, c_2, \ldots, c_k\} \forall l \in \{1, \ldots, n\} \)

2: Output: A set of cluster assignments \( C_l = \{w_1, w_2, \ldots, w_{l_i}\} \forall l \in \{1, \ldots, n\} \)

3: for \( l = 1 \to n \) do
4: for \( i = 1 \to n_l \) do
5: \( j = \arg\min_i (W_{li} - C_{lj})^2 \)
6: \( C_{lj} \leftarrow C_{lj} \cup \{W_{li}\} \)
7: return \( C \)
Appendix C

Chapter 5 - Supplemental material

C.1 Details on neural network experiments

C.1.1 Matrix preprocessing and convolutional layers

Before benchmarking the quantized weight matrices we applied the following pre-processing steps:

Matrix decomposition

After the quantization step it may well be that the 0 value is not included in the set of values and/or that it’s not the most frequent value in the matrix. Therefore, we decompose the matrix into $W_q = \hat{W} + \omega_{\text{max}}$, where $\hat{W}$ is the unit matrix whose elements are equal to 1 and $\omega_{\text{max}}$ is the element that appears most frequently in matrix $W_q$. Consequently, $\hat{W}$ is a matrix with 0 as it’s most frequent element. Moreover, when performing the dot product with an input vector $x \in \mathbb{R}^n$, we only incur the additional cost of adding the constant value $c_{\text{out}} = \omega_{\text{max}} \sum x_i$ to all the elements of the output vector. The cost of this additional operation is effectively of the order of $n$ additions and 1 multiplication for the entire dot product operation, which is negligible as long as the number of rows is sufficiently large.

Convolution layers

A convolution operation can be performed by a matrix-matrix dot product operation. The weight tensor containing the filter values would be represented as a $(F_n \times (n_{ch}m_Fn_F))$-dimensional matrix, where $F_n$ is the number of filters of the layer, $n_{ch}$ the number of channels, and $(m_F, n_F)$ the height/width of the filters. Hence, the convolution matrix would perform a dot product operation with an $(n_{ch}m_Fn_F \times n_p)$-dimensional matrix, that contains all the patches $n_p$ of the input image as column vectors.

Hence, in our experiments, we reshaped the weight tensors of the convolutional layers into their respective matrix forms and tested their storage requirements and dot product complexity by performing a simple matrix-vector dot product, but weighted the results by the respective number of patches $n_p$ that would have been used at each layer.

C.1.2 Dot product pseudocodes

Algorithms 10, 11 and 12 show the pseudocodes of the dot product algorithm of the CSR, CER and CSER data structures.

For the dense algorithm, we implemented the standard three loop nest Algorithm 9. We used the programming language Python in all our experiments.
Algorithm 9 Dense dot product

1: procedure DOT\_dense(A, X)
2: \( M, N \leftarrow \text{dim}(A) \)
3: \( N, L \leftarrow \text{dim}(X) \)
4: \( Y = 0 \in \mathbb{R}^{M \times L} \)
5: for \( l = 0 < L \) do
6: \hspace{1em} for \( m = 0 < M \) do
7: \hspace{2em} \( y = 0 \)
8: \hspace{2em} for \( n = 0 < N \) do
9: \hspace{3em} \( y \leftarrow y + A[m, n] \ast X[n, l] \)
10: \hspace{2em} \( Y[m, l] \leftarrow y \)
11: \hspace{1em} return \( Y \)

Algorithm 10 CSR dot product

1: procedure DOT\_csr(A, X)
2: \( \Omega, \text{colI, rowPtr} \leftarrow A \)
3: \( N, L \leftarrow \text{dim}(X) \)
4: \( Y = 0 \in \mathbb{R}^{M \times L} \)
5: for \( l \leq L \) do
6: \hspace{1em} for \( r_{idx} = 1 < \text{len(rowPtr)} \) do
7: \hspace{2em} \( r_{\text{start}} \leftarrow \text{rowPtr}[r_{idx} - 1] \)
8: \hspace{2em} \( r_{\text{end}} \leftarrow \text{rowPtr}[r_{idx}] \)
9: \hspace{2em} \( y = 0 \)
10: \hspace{2em} for \( i = r_{\text{start}} < r_{\text{end}} \) do
11: \hspace{3em} \( I \leftarrow \text{colI}[i] \)
12: \hspace{3em} \( y \leftarrow y + \Omega[I] \ast X[I, l] \)
13: \hspace{2em} \( Y[r_{idx} - 1, l] \leftarrow y \)
14: \hspace{1em} return \( Y \)
Algorithm 11 CER dot product

1: procedure DOT$_{cer}(A,X)$
2: \[ \Omega, \text{colI}, w\text{Ptr}, \text{rowPtr} \leftarrow A \]
3: \[ N, L \leftarrow \text{dim}(X) \]
4: \[ Y = 0 \in \mathbb{R}^{M \times L} \]
5: \[ \textbf{for } l \leq L \textbf{ do} \]
6: \[ r_{\text{start}} = 0 \]
7: \[ w_{\text{start}} = 0 \]
8: \[ \textbf{for } r_{\text{idx}} = 1 < \text{len}(\text{rowPtr}) \textbf{ do} \]
9: \[ r_{\text{end}} \leftarrow \text{rowPtr}[r_{\text{idx}}] \]
10: \[ y = 0 \]
11: \[ w_{\text{count}} = 1 \]
12: \[ \textbf{for } w_{\text{idx}} = r_{\text{start}} + 1 < r_{\text{end}} + 1 \textbf{ do} \]
13: \[ w_{\text{end}} \leftarrow \text{wPtr}[w_{\text{idx}}] \]
14: \[ y' = 0 \]
15: \[ \textbf{for } i = w_{\text{start}} < w_{\text{end}} \textbf{ do} \]
16: \[ I \leftarrow \text{colI}[i] \]
17: \[ y' \leftarrow y' + X[I, l] \]
18: \[ \textbf{if } w_{\text{start}} + 1 = w_{\text{end}} \textbf{ then} \]
19: \[ y \leftarrow y + y' \times \Omega[w_{\text{count}}] \]
20: \[ w_{\text{count}} \leftarrow w_{\text{count}} + 1 \]
21: \[ w_{\text{start}} \leftarrow w_{\text{end}} \]
22: \[ r_{\text{start}} \leftarrow r_{\text{end}} \]
23: \[ Y[r_{\text{idx}} - 1, l] \leftarrow y \]
24: return $Y$
Algorithm 12 CSER dot product

1: procedure DOT\textsubscript{cser}\(A,X\)
2: \(\Omega, \text{coll}, wI, wPtr, \text{rowPtr} \leftarrow A\)
3: \(N, L \leftarrow \text{dim}(X)\)
4: \(Y = 0 \in \mathbb{R}^{M \times L}\)
5: for \(l \leq L\) do
6: \(r\_\text{start} = 0\)
7: \(w\_\text{start} = 0\)
8: \(w\_\text{count} = 0\)
9: for \(r\_\text{idx} = 1 < \text{len}(	ext{rowPtr})\) do
10: \(r\_\text{end} \leftarrow \text{rowPtr}[r\_\text{idx}]\)
11: \(y = 0\)
12: for \(w\_\text{idx} = r\_\text{start} + 1 < r\_\text{end} + 1\) do
13: \(w\_\text{end} \leftarrow \text{wPtr}[w\_\text{idx}]\)
14: \(y' = 0\)
15: for \(i = w\_\text{start} < w\_\text{end}\) do
16: \(I \leftarrow \text{coll}[i]\)
17: \(y' \leftarrow y' + X[I,l]\)
18: \(y \leftarrow y + y' * \Omega[wI[w\_\text{count}]]\)
19: \(w\_\text{count} \leftarrow w\_\text{count} + 1\)
20: \(w\_\text{start} \leftarrow w\_\text{end}\)
21: \(r\_\text{start} \leftarrow r\_\text{end}\)
22: \(Y[r\_\text{idx} - 1,l] \leftarrow y\)
23: return \(Y\)

C.2 Proof of theorems

Theorem 2

The CER data structure represents any matrix via four arrays, which respectively contain: \(\Omega : K\), \(\text{coll} : N - #(0)\), \(w\_\text{Ptr} : \sum_{r=0}^{m} \bar{k}_r + \bar{\tilde{k}}_r\), \(\text{row\_Ptr} : m\) entries, where \(K\) denotes the number of unique elements appearing in the matrix, \(N\) the total number of elements, \#(0) the total number of zero elements, \(m\) the row dimension and finally, \(\bar{k}_r\) the number of shared elements that appeared at row \(r\) (excluding the 0) and \(\bar{\tilde{k}}_r\) the number of redundant padding entries needed to communicate at row \(r\).

Hence, by multiplying each array with the respective element bit size and dividing by the total number of elements we get

\[
\frac{K b_\Omega}{N} + \left(1 - \frac{#(0)}{N}\right) b_1 + \frac{1}{N} \left(\sum_{r=0}^{m} \bar{k}_r + \bar{\tilde{k}}_r\right) b_1 + \frac{1}{n} b_1
\]

where \(b_\Omega\) and \(b_1\) are the bit sizes of the matrix elements and the indices respectively.

With \(p_0 = \frac{#(0)}{N}\) and \(\bar{k} + \bar{\tilde{k}} = \frac{1}{m} \sum_{r=0}^{m} \bar{k}_r + \bar{\tilde{k}}_r\), we get equation (9).

The cost of the respective dot product algorithm can be estimated by calculating the cost of each line of Algorithm 12. To recall, we denoted with \(\sigma(b)\) the cost of performing a summation operation, which involved \(b\) bits. \(\mu(b)\) the cost of a multiplication. \(\gamma(b)\) the cost of a read and \(\delta(b)\) of a write operation into memory. We further denoted with \(c_{\text{oth}}\) the cost of performing other types of operations. Moreover, assume an input vector (that is, \(L = 1\)), since the result can be trivially extended to input matrices of arbitrary size. Thus, Algorithm 11 requires: from line
C.2. Proof of theorems

2) - 7) we assume a cost of $c_{\text{oth}}$, 8) $mc_{\text{oth}}$, 9) $m\gamma(b_1)$, 10) $mc_{\text{oth}}$, 11) $mc_{\text{oth}}$, 12) $m(k + \tilde{k})c_{\text{oth}}$, 13) $m(k + \tilde{k})\gamma(b_1)$, 14) $m(k + \tilde{k})c_{\text{oth}}$, 15) $N(1 - p_0)c_{\text{oth}}$, 16) $N(1 - p_0)\gamma(b_1)$, 17) $N(1 - p_0)(\gamma(b_a) + \sigma(b_a))$, 18) $mk\gamma(b_\Omega) + \mu(b_o) + \sigma(b_o) - \sigma(b_a)$, 19) $m(k + \tilde{k})c_{\text{oth}}$, 20) $m(k + \tilde{k})c_{\text{oth}}$, 21) $mc_{\text{oth}}$, 22) $m\delta(b_o)$; where $b_\Omega$, $b_1$ and $b_o$ are the bit sizes of the matrix elements, the indices and output vector element respectively. Hence, adding up all above costs and replacing $c_a$ and $c_\Omega$ as in equations (5) and (6), we can get the total cost of $c_{\text{oth}} + m(\gamma(b_1) + \delta(b_o) + 3c_{\text{oth}}) + mk\gamma(b_\Omega) + \mu(b_o) + \sigma(b_o) + \sigma(b_a) + N(1 - p_0)(c_a + c_{\text{oth}})$. It is fair to assume that the cost $c_{\text{oth}}$ is negligible compared to the rest for highly optimized algorithms. Indeed, Figures 8, 7 and 9 show that cost of these operations contribute very little to the total cost of the algorithm. Hence, we can assume the ideal cost of the algorithm to be equal to the above expression with $c_{\text{oth}} = 0$ (which corresponds to equation (10)).

□

Theorem 3

Analogously, we can follow the same line of arguments. Namely, each array in the CSER data structure contains: $\Omega : K, \text{col} : N - \#(0), wI : \sum_{r=0}^{m}\tilde{K}_r, wPtr : \sum_{r=0}^{m}\tilde{k}_r, rowPtr : n$ entries. Consequently, by adding those terms, multiplying by their bit size and dividing by the total number of elements $N$ we recover equation (11).

Each line of Algorithm 12 induces a cost of: form line 2) - 8) we assume a cost of $c_{\text{oth}}$, 9) $mc_{\text{oth}}$, 10) $m\gamma(b_1)$, 11) $mc_{\text{oth}}$, 12) $mk\gamma(b_1)$, 13) $mk\gamma(b_1)$, 14) $mk\gamma(b_1)$, 15) $N(1 - p_0)c_{\text{oth}}$, 16) $N(1 - p_0)\gamma(b_1)$, 17) $N(1 - p_0)(\gamma(b_a) + \sigma(b_a))$, 18) $mk\gamma(b_\Omega) + \mu(b_o) + \sigma(b_o) - \sigma(b_a)$, 19) $mk\gamma(b_\Omega) + \gamma(b_1) + \mu(b_o) + \sigma(b_o) - \sigma(b_a) + N(1 - p_0)(c_a + c_{\text{oth}})$.

Again, adding up all terms and replacing with $c_a$ and $c_\Omega$ then we get the total cost of $c_{\text{oth}} + m(\gamma(b_1) + \delta(b_o) + 3c_{\text{oth}}) + mk\gamma(b_\Omega) + \gamma(b_1) + 4c_{\text{oth}} + N(1 - p_0)(c_a + c_{\text{oth}})$ and with $c_{\text{oth}} = 0$ we recover equation (12).

□
Appendix D

Chapter 6 - Supplemental material

D.1 Experiments - A more detailed description and further results

All experiments were conducted using the PyTorch deep learning framework, version 1.5. Small-scale image classification tasks, Keyword Spotting and Hand Gesture Recognition tasks were executed on a single NVIDIA TITAN V GPU with CUDA Version 10.1. For ImageNet experiments two (ResNets) or three (EfficientNet-B0) Tesla V100 GPUs with CUDA version 10.2 were used.

The algorithm usually converges within 20 to 30 epochs of trained quantization. Retraining the non-quantized layers, i.e. BatchNorm- or Bias-Layers, can compensate for drops in accuracy caused by quantization and works especially well for EfficientNets and custom MLPs for hand gesture recognition.

For a fair comparison and transparent benchmarking we used publicly available pre-trained networks. ResNet-50 and -34 come from the torchvision model zoo \(^1\), EfficientNet-B0 from \(^2\) and ResNet-20 from \(^3\).

For better comparison we do not quantize the input and output layers of the ResNets. Apart from that, all other models are quantized completely to 4 bit, except from the BatchNorm and bias parameters.

In the experiments section we distinguish between hardware-conform and non-conform models. Conform models include only fully connected layers with up to 512 input/output features. Optionally, BatchNorm layers are allowed which can result in accuracy gains.

To cover a variety of possible application showcases with the conform models, we deployed several models solving classification tasks for audio, image, biomedical and sensor data. For this approach we implemented custom and well-known MLPs. All our custom MLPs use Dropout (50\%) between the last hidden layer and the output layer. They were trained for 100 epochs using stochastic gradient descent (SGD) optimization with a momentum of 0.9 and a cosine annealing learning rate schedule without restarts. The initial learning rate is 0.01. To benchmark our quantization algorithm we also used non-conform models, i.e. the EfficientNet-B0, which we have not trained ourselves but obtained from the above mentioned sources.

---

D.1.1 Hand Gesture Recognition

The authors in (Georgi, Amma, and Schultz, 2015) collected Inertial Measurement Unit (IMU) and electromyogram (EMG) readings from 5 different subjects in 5 different sessions in order to capture 12 defined hand gestures. The IMU unit consists of a 3D accelerometer and a 3D gyroscope, providing 6 IMU data channels. The EMG data is recorded from 16 electrodes placed on the forearm. Different from (Georgi, Amma, and Schultz, 2015), we deploy a small MLP to solve the classification task. It clearly outperforms the proposed Hidden Markov Model which achieves a mean accuracy of 74.3% for person-independent hand gesture recognition. Our proposed 4-layer deep MLP, which we named MLP-HR, achieves a person-independent mean accuracy of 84.0% (with accuracies of 0.885, 0.814, 0.892, 0.840 and 0.770 utilizing recordings of subjects A, B, C, D and E as test samples). Quantizing all network layers to 4 bit with the FantastIC4 algorithm is possible with almost no drop in accuracy. The model consists of an input layer, two hidden layers and an output layer with 512, 256, 128 and 12 output features, where a BatchNorm layer follows each fully connected layer. In general, the data preprocessing follows (Georgi, Amma, and Schultz, 2015), that is we z-normalize both signals, IMU and EMG, and, for feature generation, calculate the standard deviation (std) of the EMG signal and the IMU signal plus the average value (mean) of the IMU signal in a moving window procedure. Thus, the feature vector per window slide is of length 28, including 12 elements for 6 IMU channels (std and mean features) as well as 16 elements for the 16 EMG electrodes using mean features only. The stride of the moving window is half of the window length and we move it 10 times, generating a feature vector of length 308 as input for the input layer of our neural network. The data corpus is publicly available 4.

D.1.2 Keyword Spotting (KWS)

Keyword spotting is a classification task designed to support short speech command recognition. The Google Speech Commands dataset consists of 105,829 utterances of 35 words recorded from 2,618 speakers. The standard is to discriminate ten words “Yes”, “No”, “Up”, “Down”, “Left”, “Right”, “On”, “Off”, “Stop”, and “Go”, and adding two additional labels, one for “Unknown Words”, and another for “Silence” (no speech detected) (Warden, 2018). There are no overlapping speakers between the train, test and validation sets. Following the official Tensorflow example code for training 5, we implemented the corresponding data augmentation with PyTorch’s torchaudio package. It includes randomly adding background noise with a probability of 80% and time shifting the audio [−100, 100] ms with a probability of 50%. To generate features, the audio is transformed to MFCC fingerprints. We use 15 bins and a window length of 2000 ms. We deploy a MLP consisting of an input layer, five hidden layers and an output layer featuring 512, 512, 256, 256, 128, 128 and 12 output features, respectively.

Our model, named MLP-GSC, achieves a classification accuracy of 91.0% which outperforms the default CNN model (88.2%) in the TensorFlow example code mentioned in (Warden, 2018). The FantastIC4 4 bit quantization has a regularizing effect on the full-precision MLP and further improves the classification accuracy to 91.35%, while introducing 60% sparsity. The authors in (Zhang et al., 2017) show that for the
Google Speech Commands dataset CNNs and especially RNNs usually achieve better accuracies than MLPs. Still, our proposed model yields an comparable accuracy to their proposed CNN and outperforms their 8 bit quantized MLP (88.91%). For another comparison, (Liu et al., 2019) quantized their network composed by three convolution layers and two fully-connected layers to 7 bit using 8 bit activations, and achieve an accuracy of 90.82%.

### D.1.3 Image Classification

For the subsequent image classification tasks we use standard data preprocessing, i.e. normalization and, for CIFAR-10 and ImageNet, random horizontal flipping and cropping.

**Small-scale Image Classification**

For small-scale image classification we utilized two neural networks, one MLP which would fit into our proposed accelerator, LeNet-300-100, and one CNN (ResNet-20). CIFAR-10 (Krizhevsky, 2009) is a dataset consisting of natural images with a resolution of $32 \times 32$ pixels. It contains 10 classes. The train and test sets contain 50,000 and 10,000 images. MNIST (LeCun and Cortes, 2010) is drawn from 10 classes where each class refers to a handwritten digit (0-9). The dataset contains 60,000 training images and 10,000 test images with a resolution of $28 \times 28$ pixels.

**Large-scale Image Classification**

To benchmark our quantization algorithm with ImageNet we deployed EfficientNet-B0, ResNet-50, and -34 networks. The ImageNet (Russakovsky et al., 2015) dataset is a large-scale dataset containing 1.2 million training images and 50,000 test images of 1000 classes. The resolution of the image data is various and in the range of several hundred pixels. We crop the ImageNet data in all experiments to $224 \times 224$ pixels.

### D.1.4 Further and more extensive results

In table D.1 we provide more data regarding our experimental results generated from training the above described DNN models. More concretely, we provide data regarding further models in the Pareto-optimal front, and indicate also the achieved sparsity ratios for better comparison with previous work. Moreover, we also include information regarding the bit-precision applied to the activation values for the benchmarking.

### D.2 Ablation studies

#### D.2.1 Increased capacity of FantastIC4 models

In the following we conduct experimental ablation studies that motivated the FantastIC4’s particular hardware design.

**Sensitivity of activation values vs DNN parameters**

In figure D.1 we show experimental evidence that, indeed, the activations of DNN models are more sensitive to quantization than the weight parameters. These results
motivated the support of mixed-precision values in FantastIC4’s hardware architecture.

**Benefit from learning cluster centers vs fixed cluster centers**

The benefit from allowing learnable $w_i$ instead of fixed $w_i$ being powers of 2, is depicted in figure D.2. The experiments cover Google Speech Commands, Hand Gesture Recognition and ImageNet datasets. As a hyperparameter, different $\lambda$ were used to control the intensity of the entropy constraint and thus, as shown in figure 6.9 in chapter 6, the power efficiency of the resulting network.

When enabling the fine-tuning of the $w_i$ cluster centers with the Straight-Through Estimator, we obtained better performing neural networks. Especially networks with higher compression rates seem to benefit from the trained cluster centers. Both experiments, i.e. trained $w_i$ and fixed $w_i$, start from an identical 4-bit integer-aligned distribution.

By restricting $w_i$ to powers of 2, multiplications could be eliminated. However, it should also be considered that BatchNorm parameters are ubiquitous in neural networks and that they require multiplication operations for each output activation. By merging $w_i$ with the BatchNorm parameters, we reduce the total number of multiplications to only 4 per output activation and thus intend to find a sweet spot in terms of computational complexity and model performance.
D.2. Ablation studies

Figure D.2: From left to right: Quantization process of MLP-GSC, MLP-HGR, EfficientNet-B0 solving Google Speech Commands, Hand Gesture Recognition and ImageNet. The curves compare the trained quantization of learned $w_l$ vs int4-aligned $w_l$ (powers of 2). $\lambda$ controls the entropy and thus the overall network power efficiency ©2021 IEEE.
Table D.1: Comparison of the FantastIC4-quantization approach vs state-of-the-art 4-bit quantization techniques. Unless otherwise specified, all approaches quantize all network layers, including input- and output-layers, excluding batch normalization- and bias-parameters.

<table>
<thead>
<tr>
<th>Model</th>
<th>Quant.</th>
<th>Acc. (%)</th>
<th>Acc. drop</th>
<th>$\frac{W=0}{W}$ (%)</th>
<th>Size (MB)</th>
<th>CR</th>
<th>CSR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ImageNet</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EfficientNet-B0</td>
<td>W4A16</td>
<td>75.01</td>
<td>-1.42</td>
<td>22.23</td>
<td>21.15</td>
<td>7.62</td>
<td>3.31</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>74.08</td>
<td>-2.35</td>
<td>34.57</td>
<td>21.15</td>
<td>8.25</td>
<td>3.91</td>
</tr>
<tr>
<td></td>
<td>W4A4</td>
<td>73.80</td>
<td>-2.30</td>
<td>0</td>
<td>21.15</td>
<td>7.48</td>
<td>2.59</td>
</tr>
<tr>
<td>LSQ+ Bhagat et al., 2020</td>
<td>W4A4</td>
<td>73.80</td>
<td>-2.30</td>
<td>0</td>
<td>21.15</td>
<td>7.48</td>
<td>2.59</td>
</tr>
<tr>
<td><strong>ResNet-50</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>75.66</td>
<td>-0.49</td>
<td>24.61</td>
<td>102.23</td>
<td>8.21</td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>75.29</td>
<td>-0.86</td>
<td>41.57</td>
<td>102.23</td>
<td>9.97</td>
<td>4.50</td>
</tr>
<tr>
<td></td>
<td>W4A8</td>
<td>75.62</td>
<td>-0.51</td>
<td>0</td>
<td>102.23</td>
<td>7.86</td>
<td>2.64</td>
</tr>
<tr>
<td>PWLQ Fang et al., 2020†</td>
<td>W4A32</td>
<td>74.98</td>
<td>-1.04</td>
<td>0</td>
<td>102.23</td>
<td>7.94</td>
<td>2.64</td>
</tr>
<tr>
<td>MMSE Choukroun et al., 2019</td>
<td>W4A32</td>
<td>74.98</td>
<td>-1.04</td>
<td>0</td>
<td>102.23</td>
<td>7.94</td>
<td>2.64</td>
</tr>
<tr>
<td>KURE Shkolnik et al., 2020†</td>
<td>W4A32</td>
<td>75.60</td>
<td>-0.70</td>
<td>0</td>
<td>102.23</td>
<td>7.88</td>
<td>2.64</td>
</tr>
<tr>
<td><strong>ResNet-34</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>72.98</td>
<td>-0.32</td>
<td>38.98</td>
<td>87.19</td>
<td>7.80</td>
<td>4.32</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>72.86</td>
<td>-0.44</td>
<td>39.67</td>
<td>87.19</td>
<td>9.30</td>
<td>4.37</td>
</tr>
<tr>
<td></td>
<td>W4A4</td>
<td>73.70</td>
<td>-0.00</td>
<td>0</td>
<td>87.19</td>
<td>6.82</td>
<td>2.65</td>
</tr>
<tr>
<td>QIL Jung et al., 2019†</td>
<td>W4A4</td>
<td>72.76</td>
<td>-1.04</td>
<td>0</td>
<td>87.19</td>
<td>6.82</td>
<td>2.65</td>
</tr>
<tr>
<td>DSQ Gong et al., 2019†</td>
<td>W4A4</td>
<td>72.76</td>
<td>-1.04</td>
<td>0</td>
<td>87.19</td>
<td>6.82</td>
<td>2.65</td>
</tr>
<tr>
<td><strong>CIFAR-10</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResNet-20</td>
<td>W4A16</td>
<td>91.60</td>
<td>-0.07</td>
<td>33.60</td>
<td>1.08</td>
<td>8.43</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>91.15</td>
<td>-0.52</td>
<td>78.07</td>
<td>1.08</td>
<td>16.23</td>
<td>11.31</td>
</tr>
<tr>
<td>SLB Yang et al., 2020†</td>
<td>W4A32</td>
<td>92.10</td>
<td>-0.00</td>
<td>0</td>
<td>1.08</td>
<td>7.64</td>
<td>2.62</td>
</tr>
<tr>
<td>GWS Zhong et al., 2020†</td>
<td>W4A4</td>
<td>91.46</td>
<td>-0.74</td>
<td>0</td>
<td>1.08</td>
<td>7.72</td>
<td>2.62</td>
</tr>
<tr>
<td><strong>MNIST</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeNet-300-100</td>
<td>W4A16</td>
<td>98.63</td>
<td>-0.07</td>
<td>65.96</td>
<td>1.07</td>
<td>13.31</td>
<td>7.62</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>98.16</td>
<td>-0.54</td>
<td>86.79</td>
<td>1.07</td>
<td>29.31</td>
<td>19.81</td>
</tr>
<tr>
<td><strong>Google Speech Commands</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FantastIC4-Net</td>
<td>W4A16</td>
<td>91.19</td>
<td>+0.19</td>
<td>52.92</td>
<td>2.57</td>
<td>10.88</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>90.99</td>
<td>-0.01</td>
<td>63.02</td>
<td>2.57</td>
<td>12.58</td>
<td>7.02</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>90.41</td>
<td>-0.59</td>
<td>67.65</td>
<td>2.57</td>
<td>13.59</td>
<td>7.99</td>
</tr>
<tr>
<td>HE Zhang et al., 2017</td>
<td>W4A16</td>
<td>86.70</td>
<td>-0.00</td>
<td>0</td>
<td>0.20</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td><strong>Hand Gesture Recognition</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FantastIC4-Net</td>
<td>W4A16</td>
<td>88.33</td>
<td>-0.17</td>
<td>34.23</td>
<td>1.30</td>
<td>8.51</td>
<td>3.96</td>
</tr>
<tr>
<td></td>
<td>W4A16</td>
<td>87.82</td>
<td>-1.28</td>
<td>69.70</td>
<td>1.30</td>
<td>13.57</td>
<td>8.35</td>
</tr>
</tbody>
</table>

a Quantization scheme: WxAx indicates a quantization of weights and activations to x bits. Sparse, measured as the percentage of zero-valued parameters in the whole neural network. 

b Compression ratio defined as the ratio of the full-precision model size to the quantized model size, where FantastIC4 stores each layer in its optimal format which is either CSR, bitmask format or the trivial 4bit dense format. 

c Compression ratio defined as the ratio of the full-precision model size to the quantized model size, where each layer is stored in CSR format. 

† QIL and DSQ use full-precision (32-bit) for the first and last layer, PWLQ and SLB use full-precision for the first layer and KURE and GWS provide no information about first/last layer quantization. 

Our ResNet-34/10 benchmark has 32-bit input- and output-layers and the ResNet-34 benchmark a 32-bit input layer. ‡ MMSE reported their achieved CR. For KURE and GWS we assume that they quantized all convolution and fully-connected layers.
Bibliography


Fang, Jun et al. (2020). “Post-Training Piecewise Linear Quantization for Deep Neural Networks”. In: ECCV.


Jouppi, N. P and et.al (2017). “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *44th International Symposium on Computer Architecture (ISCA)*. ISSN: 00280836. DOI: 10.1038/264079a0.

Jung, Sangil et al. (2019). “Learning to Quantize Deep Networks by Optimizing Quantization Intervals With Task Loss”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.


Lian, Xiaocong et al. (2019). “High-performance fpga-based cnn accelerator with block-floating-point arithmetic”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8, pp. 1874–1885. ISSN: 15579999. DOI: 10.1109/TVLSI.2019.2913958.


Sattler, Felix, Klaus-Robert Müller, and Wojciech Samek (2020). “Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints”. In: *IEEE Transactions on Neural Networks and Learning Systems*.


Sattler, Felix et al. (2019). “Sparse Binary Compression: Towards Distributed Deep Learning with minimal Communication”. In: *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. DOI: 10.1109/IJCNN.2019.8852172.


Shivapakash, Suhas et al. (2020). “A Power Efficient Multi-Bit Accelerator for Memory Prohibitive Deep Neural Networks”. In: IEEE Circuits and System Conference, Accepted paper.


Silver, David et al. (2016). “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: Nature 529.7587, pp. 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.


Bibliography


Won, Dong-Ok, Klaus-Robert Müller, and Seong-Whan Lee (2020). “An adaptive deep reinforcement learning framework enables curling robots with human-like performance in real-world conditions”. In: Science Robotics 5.46. DOI: 10.1126/scirobots.abb9764. eprint: [https://robotics.sciencemag.org/content/5/46/eabb9764.full.pdf](https://robotics.sciencemag.org/content/5/46/eabb9764.full.pdf). URL: [https://robotics.sciencemag.org/content/5/46/eabb9764](https://robotics.sciencemag.org/content/5/46/eabb9764).


Zhu, Michael H. and Suyog Gupta (2018). To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. URL: https://openreview.net/forum?id=S1N69AT-.